TRANSPUTER IMPLEMENTATION OF SYSTOLIC ARRAYS

A Thesis Submitted

In partial Fulfilment of the Requirements

For the Degree of

MASTER OF TECHNOLOGY

by

M.K.V.SUBBA RAO

to the

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

April, 1992.

Dedicated to

My parents

EE-1992-M-RAO-TRA

ACKNOWLEDGEMENTS

I wish to express my deep sense of gratitude and reverence to Dr. M.U.Siddiqi, my thesis supervisor, for his valuable guidance and constant encouragement throughout the course of this work. His instructions and suggestions have proved to be of immense help at every stage.

I am grateful to Mr. Hari Prasad for his valuable help and advise during the course of the work.

I am very much thankful to Shreenivas Galgali for his cooperation and company during my work.

Udaya and Madhu helped me a lot during my report preparation. My special thanks to them. Discussions with Venkatesh have been very valuable. I am grateful for his constructive criticism.

My special thanks to Venugopal for the enthusiasm and interest he has shown in my work.

I wish to extend my thanks to Alok, Peeyush, and Mahajan for their company and cooperation.

I must thank my friends Vissu, Peri, Venku, Ravi, Sankar, Ravindra, Siva, Dandam, Venu, and Umakant who were enjoyable and memorable company during my stay in Kanpur.

ABSTRACT

The thesis aims at implementing systolic signal processing algorithms on a transputer array. Matrix multiplication, convolution, Fast Fourier transform, sorting, and Fast Walsh–Hadamard algorithms are selected for parallelization. The graph based systolization procedure is chosen for deriving systolic algorithms from their sequential versions. The algorithm design phases consist of the selection of projection and scheduling vectors, mapping the computation onto a processor array and problem partitioning. The partitioned algorithms are implemented on 2, 4, and 8 transputer arrays. All the program coding is done in Occam, the native language for the transputers. The program development, testing and debugging is done under the Transputer Development System environment. The performance of the systolic structures is estimated.

CERTIFICATE

This is to certify that the work entitled "TRANSPUTER IMPLEMENTATION OF SYSTOLIC ARRAYS" has been carrid out by M.K.V.Subba Rao under my supervision and has not been submitted elsewhere for a degree.

[illegible]
(M.U.Siddiqi) 23-4-92
Professor
Department of Electrical Engineering
IIT KANPUR.

April, 1992.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1. Scope of the work :

Parallel and distributed computation is currently an area of intense research activity, motivated by a variety of factors. There has always been a need for the solution of very large computational problems, but it is only recently that technological advances have raised the possibility of parallel computation and have made the solution of such problem possible. Accordingly, the development of parallel and distributed algorithms is guided by the interplay between old and new computational needs on one hand, and the technological progress on the other.

Until the advent of VLSI most signal processing algorithms were implemented in software on single processor systems, or special purpose hardware systems were designed using standard LSI chips. Now it is possible to consider algorithm implementation on either an array of parallel processors in which each processor is a Von Neumann machine or via an array of simple processors fabricated on a single chip or chip set. The first case is exemplified by the TRANSPUTER, while the second approach is typified by SYSTOLIC arrays.

The fundamental signal processing problem inherent in either case is how to proceed from an algorithm to a hardware/software design capable of being implemented on parallel architectures. To fully exploit the potentials that VLSI offers to signal processing it is necessary to place the design and evaluation of regular array architectures on a more systematic basis. Several approaches have recently been proposed for designing regular arrays based on graph theoretic methods. In this thesis, we are using the design proposed by H.T.Kung [1] (which uses the data interleaving and cut set delay transformation) to obtain signal flow graphs of architectures capable of being implemented as SYSTOLIC arrays. The aim of this Thesis is to implement systolic arrays on transputers using the OCCAM, the native programming language for the transputers. The transputer is used as a systolic cell and the cell data path is provided by the transputer links.

There are several reasons for using transputer as a systolic cell. First, the processors require minimum external hardware support for reasonably good interprocessor communication. Secondly, the arithmetic processor itself is very powerful compared to the conventional microprocessors (it is based on the RISC philosophy). The array can be easily expanded in size easily; adding a new processor to the array is much more simpler than that with any other multiprocessor system. It has a highly structured native language, Occam, which systematizes the code development phase. Last but not the least, it is cost effective compared with any available supercomputing system available in the market.

2. Implementation:

The phases in systolic array implementation are as follows.

1. An algorithms is systolized from its sequential version using standard graph based methods.
2. The systolic algorithm is coded in occam algebra as a number of communication parallel processes and tested on a single processor machine simulating the effect of pseudo parallelism.
3. The algorithm may now be run on a processor array in which each transputer implements one or more of the Occam processes.

3. Organization of the Thesis :

The organization of the Thesis is as follows :

Chapter 2 is a brief survey of the state of the art in parallel processing array architectures. It also provides an introduction to systolic arrays and explain their salient features.

Chapter 3 describes the standard design methodologies for systolic algorithms using dependence analysis, namely the canonical mapping and generalized mapping methodologies. Partitioning of a larger problem to suit an array of smaller size is also discussed.

A few key signal processing algorithms are systolized in Chapter 4 using the methods described in Chapter 3. Dependence graph, Projection vectors, Scheduling of processors, and finally, the partitioning of the problem are described in detail.

Chapter 5 describes the architecture of the transputer and its native language occam. For writing efficient parallel programs on transputers, a good working knowledge, and understanding of the special features of the processor are absolutely necessary. Key issues in the design of parallel algorithms specially for transputers are discussed in this chapter.

Chapter 6 describes the hardware setup used for the implementation of the algorithms. It also discusses some of the issues (e.g., configuration, array synchronization and code development issues) regarding implementation of systolic algorithms on transputer network.

Chapter 7 evaluates the performance of various algorithms respectively on a network of 2, 4 and 8 transputers (configured in a linear one dimensional systolic array fashion). The sequential version of the algorithms is also implemented on both transputer as well as the IBM PC–AT and their relative performances are compared. The performance of the transputer array is compared with that of the CMU Warp computer. The thesis is concluded in this chapter discussing the scope for further work.

CHAPTER 2

SUPER-COMPUTING WITH VLSI ARCHITECTURES

1. Introduction :

The increasing demands of speed and performance in in modern signal and image processing applications necessitate a revolutionary super–computing technology. The availability of low–cost, high–density, high–speed, very large scale integration devices and emerging computer aided design (CAD) facilities presages a major breakthrough in the design of massively parallel processors. In particular, VLSI microelectronics technology has inspired many innovative designs in array processor architectures. In the last decade, there has been a dramatic worldwide growth in research and development efforts on mapping various signal and image processing applications onto such VLSI architectures.

Modern signal/image processing technology depends critically on the device and architectural innovations of the computing hardware. Sequential systems will be inadequate for future real–time processing systems, and the additional computational capability available through VLSI concurrent array processors will become a necessity [1]. For example, let us examine some of the application requirements in a real–time vision processing system. The job is to recognize an object and check its geometric and physical properties against some given specifications to determine if the object is a target or not. This is a real–time application with a rate of 512 x 512 pixel image frames per millisecond. Therefore the processing speed required will be

$$10 \text{ ops/pixel} \times (512 \times 512 \text{ pixels/frame}) \times 1000 \text{ frames/s} \approx 2500 \text{ MOPs}.$$

Even for a commercial application such as video processing, the process speed required will be approximately

$$10 \text{ ops/pixel} \times (512 \times 512 \text{ pixels/frame}) \times 24 \text{ frames/s} \approx 60 \text{ MOPs}.$$

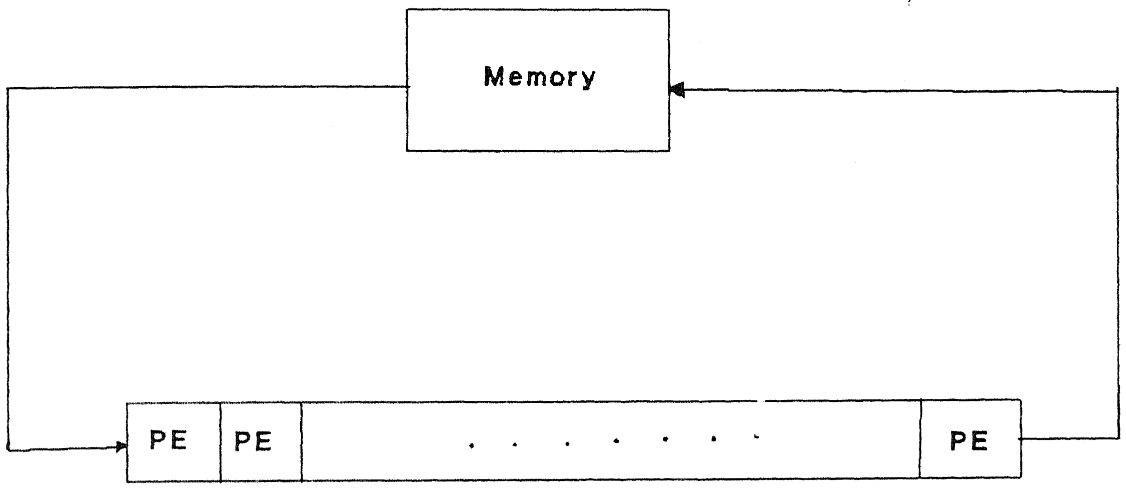


Fig. 2.1. Basic Configuration of Systolic Arrays

The general–purpose parallel computers cannot afford satisfactory processing speed due to severe system overheads. It is quickly apparent that special–purpose parallel processing architectures are indispensable.

2. Architectures :

current parallel computers can be divided into three structural classes:

- vector processors
- multiprocessor systems
- array processors

The development of these systems requires a complicated design of control units and optimized schemes for the allocation of machine resources. The third class, however, belongs to the domain of special–purpose computers, and the design of such systems requires a broad knowledge between the design of parallel computing algorithms and optimal computing hardware and software structures. The interest of the present research is focused upon the last class, since it offers a promising solution to meet real–time processing requirements. In particular, locally connected computer networks, such as systolic and wavefront arrays, are well suited to efficiently implement a major class of signal processing algorithms, due to their massive parallelism and regular data flow.

3. Systolic Array Processors :

"Systolic array processors" are a class of 'pipelined' array architectures. According to H.T.Kung [2], "***A systolic system is a network of processors which rhythmically compute and pass the date through the system***". The systolic array features the important properties of modularity, regularity, local interconnection, a high degree of pipelining, and highly synchronous multiprocessing.

The systolic array design differs from the design of the conventional Von Neumann computer in its highly pipelined computation. More precisely, once a data item is brought out from the memory, it can be used effectively at each cell it passes while being "*pumped*" from cell to cell along the array (fig. 2.1).

Conceptually, the computational tasks can be classified into two families :

1. Compute–bound computation.
2. I/O–bound computation.

In a computation, if the total number of operations is larger than total number of I/O operations, then the computation is ***compute–bound***, otherwise it is ***I/O–bound***. Speeding up the I/O–bound computation requires an increase in memory bandwidth, which is difficult in current technologies. Speeding up the compute–bound computation, however, may be accomplished by using Systolic arrays.

The main features of the systolic arrays are [1]:

- **Synchrony** : Data are rhythmically computed and passed through the network.
- **Modularity and regularity** : The array consists of modular processing elements with homogeneous interconnections. Moreover, the computational network may be extended indefinitely.
- **Spatial and temporal locality** : The array manifests a locally communicative interconnection structure,i.e., spatial locality. There is at least one unit time delay so that signal transactions from one node to next node can be completed i.e., temporal locality.
- **Pipelinability** : The array exhibits a linear rate pipelinability; i.e, it should achieve O(**M**) speedup, in terms of processing rate, where **M** is the number of processing elements (PEs). Hence the efficiency of the array is measured by the following:

$$\text{speedup} = \frac{\text{Processing time on a single processor}}{\text{Processing time on the array processor}}$$

The major factors for adopting Systolic arrays for special purpose architectures are [2]:

(a) Simple and regular design : In integrated circuit technology, the cost of computation is dropping dramatically; and at the same time the design cost grows with the complexity of the system. Hence by using simple, regular and modular design, the systems can be made to meet a wide range of applications with optimum cost.

(b) Concurrency and communication : An important factor in the potential speed of a communication system is the use of concurrency. For the special purpose systems, the concurrency depends on the underlying algorithm employed by the system. When a large number of processors work together, communication becomes significant. Therefore regular and local communication in systolic arrays is advantageous.

(c) Balancing computation with I/O : A systolic array is typically used as an attached array processor, and it receives data and outputs results through a host computer. Therefore, I/O considerations have to be taken into account in the overall performance. The ultimate performance of an array processor system is the computational rate that balances the available I/O bandwidth with the host. The important algorithmic design issue is how to arrange a computation, together with an appropriate memory structure and I/o bandwidth, so that computation time is balanced with I/O time.

The I/O problem is especially severe when the computation of a problem of large dimension is performed on a smaller array. Inevitably, it involves a partitioning problem, i.e.,computation must be decomposed. In practice this is often the case. Hence, how the problem can be decomposed to minimize I/O is also critical to the practical design of an array processor system.

CHAPTER 3

DESIGN METHODOLOGY FOR SYSTOLIC ARRAYS

3.1. Introduction:

Due to fast progress of VLSI technology, algorithm oriented array architectures appear to be effective, feasible and economic. In particular, a large class of regular computations, especially for signal and image processing, can be efficiently implemented on systolic arrays. These trends have necessitated a systematic methodology of mapping computations onto systolic arrays.

3.2. Design preliminaries:

Parallel algorithm expressions may be derived by two approaches:

1. Vectorization of sequential algorithm expressions.

2. Direct parallel algorithm expressions, such as snapshots, recursive equations, parallel codes, single assignment code, dependence graphs, and so on.

The vector processor recognizes the vectored expressions in the code and performs the operations on a vector of operands in parallel. Generally it will have vectored registers and multiple ALUs to perform the vector operations simultaneously. Detecting and analyzing the dependencies between the statements within loops is the major task in vectorization.

Since a vectorizing compiler may not be sufficiently effective in extracting the inherent concurrent (parallel and pipeline) processing, it is advantageous that a user/designer use parallel expressions to describe an algorithm in the first place. This is a key step towards an algorithm oriented array processor design. A major factor in selecting an algorithm expression is that it should express the algorithm clearly and concisely.

3.2.1 Direct expressions of parallel algorithms [1]:

3.2.1.1. Single assignment code:

A single assignment code is a form where every variable is assigned one value only during the execution of the algorithm. To transform a loop into single assignment code, the number of indices of the vectors in the loops should be increased. For example,

```
          DO 10 I = 1,4
          C(I) = 0
          DO 10 J = 1,4
          C(I) = C(I) + A(I,J) * B(J)
      10  CONTINUE
```

can be converted into single assignment form as

```
          DO 10 I = 1,4
          C(I,1) = 0
          DO 10 J = 1,4
          C(I,J+1) = C(I,J) + A(I,J) * B(J)
      10  CONTINUE
```

Each elements of the vector C will be assigned one value only. Thus this program is indeed a single assignment code.

The single assignment code is critical to the derivation of DGs.

3.2.1.2. Recursive algorithm:

A convenient and concise expression for the representation of many algorithms is to use recursive equations. The recursive equation for the above program segment is

$$c_i^{(j+1)} = c_i^{(j)} + a_i^{(j)} b_i^{(j)}$$

Where j is the recursion index, $j = 1,2,\dots,N$ and

$$c_i^{(1)} = 0$$

$$a_i^{(j)} = A(i,j)$$
$$b_i^{(j)} = B(j)$$

A recursive equation with space–time indices uses one for time and the other indices for space. By doing so, the activities of the parallel algorithms can be adequately expressed. We also note that a recursive algorithm is inherently given in a single assignment formulation.

3.2.1.3. Snapshots:

A snapshot is a description of the activities at a particular time instant. Snapshots are perhaps the most natural tool an algorithm designer can adopt to check or verify a new array algorithm.

3.2.1.4. Dependence Graph (DG):

The regular computations of an algorithm can be represented in some computational graph known as *Dependence graph* (DG). A dependence graph is a graph that shows the dependence of computation that occur in an algorithm. A DG can be considered as a graphical representation of a single assignment algorithm. The dependency relations are indicated as arcs between the corresponding variables located in the index space.

3.3. Mapping algorithms to arrays [1]:

Two mapping methodologies are proposed for mapping computational graphs on to systolic arrays.

1. Canonical mapping
2. Generalized mapping.

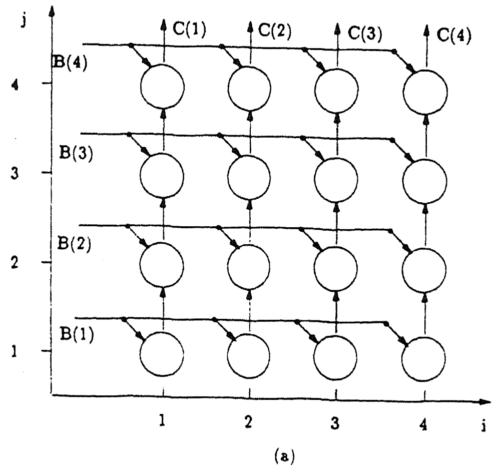


Fig. 3.1. A Typical DG With Global Communication.

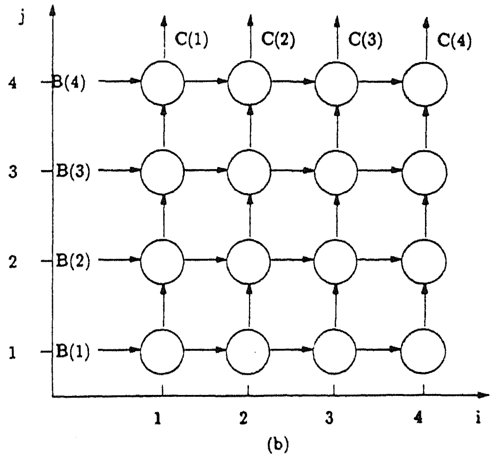


Fig. 3.2. A Typical DG With Local Communication

Both design methodologies follow the same design procedure, except for the fact that the first one is used for regular DGs and the other one for the irregular DGs.

3.3.1. Design of a DG:

There may be more than one algorithm existing for a problem having same complexity, each with its own merits and demerits. But not all algorithms are amenable for parallelization. It is the responsibility of the user to identify a suitable algorithm. Since the choice of the algorithm greatly affects the final array design, an algorithm whose dependence graph is simple, regular and locally recursive must be chosen.

To achieve maximal parallelism in an algorithm, we must carefully study the data dependencies in the computations. In the special case when the operations of a sequential algorithms have no data dependencies, they can be executed at the same time in a parallel computer. However, there is always a certain degree of dependency as indicated by the DG of the algorithm, which dictates the sequence of computation.

We must represent the algorithm in a single assignment and locally recursive forms. The space–time indices can be used to represent the algorithm in an index space. If the dependencies between the nodes in the space are shown by arcs, it is called the dependence graph representation of an algorithm. ***A graph is computable if and only if its complete DG contains no loops or cycles.*** If in a DG, all the variables are dependent on the variables on the neighboring nodes only, it is called *localized DG.* A DG is made a localized DG by replacing all the global arcs by local arcs.

The idea of local and regular DGs was explored by Karp, Miller and Winograd in their "***Systems of uniform recurrence equations***" [1]. They first used an index space display to show complete dependency of locally recursive algorithm. Along this line, some instrumental notions such as reduced DG and separating ***equitemporal hyperplanes*** and degree of parallelism are introduced. Furthermore, a graph theoretical approach and integer programming techniques for analyzing the locally recursive algorithms were also proposed.

3.3.2. Signal Flow Graph design (SFG):

An SFG is simplified graph and is close to hardware level design. Two steps are involved in the design of SFG from the DG.

- Processor assignment
- Scheduling

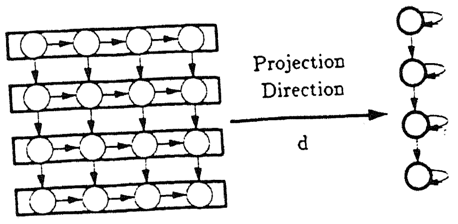
In order to derive a regular systolic array, the regularity of DG can be used to advantage. Therefore, regular assignments and scheduling attract more attention. It is common to use linear projection (in canonical mapping) in which case the nodes of the DG in a certain straight line are projected (assigned) to a PE (processing element) in the array and a linear scheduling for schedule assignment, in which case the nodes in the parallel hyperplane in the DG are scheduled to be processed at the same time step.

3.3.2.1. Processor assignment:

Since the DG of a locally recursive algorithm is very regular, the projection maps the DG onto a lower dimension lattice of points, known as "*Projection space*". Mathematically, a linear projection is often represented by a projection vector $\vec{d}$. The results of this projection are represented by an SFG. Thus an n–dimensional DG can be mapped to (n–1) dimensional processor space and 1–dimensional delay space. The delay space is related to the scheduling as explained below.

3.3.2.2. Scheduling:

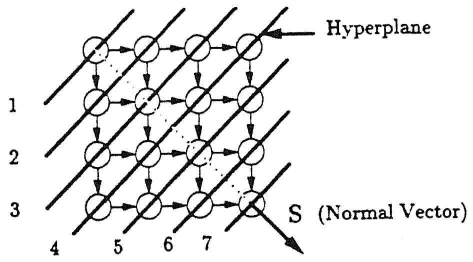
The projection should be accompanied by a scheduling scheme, which specifies the sequence of operations in all the PEs. A schedule function represents a mapping from N–dimensional index space of DG onto a 1–dimensional schedule (time) space. A linear schedule is based on a set of parallel and uniformly placed hyperplanes in the DG. These hyperplanes are called the "***equitemporal hyperplanes***". All the nodes on the same



(a)

Fig. 3.3. Linear Projection In Direction $\vec{d}$.

(Processor Assignment)



(b)

Fig. 3.4. A Linear Schedule $\vec{s}$

And Its Hyperplanes.

hyperplane must be processed at the same time. Mathematically the schedule can be represented by a (column) scheduling vector $\vec{s}$, pointing to the normal direction of the hyperplanes. For any index point i, in the DG, the time step is $\vec{s}^T i$.

The feasibility of a schedule is determined by the partial ordering and the processor assignment schemes.

Permissible linear schedules:

The necessary and sufficient conditions are,

(1) $\vec{s}^T \vec{e}>0$, for any dependence arc $\vec{e}$.

(2) $\vec{s}^T \vec{d}>0$

In short, the schedule is permissible, if and only if (a) all the dependency arcs flow in the same direction across the hyperplanes; and (b) the hyperplanes are not parallel with the projection vector $\vec{d}$. The first condition means that a causality should be enforced in a permissible scheme. The second condition implies that the nodes on an equitemporal hyperplane should not be projected on to the same PE.

If there is atleast one delay on each of the edges of the SFG, then the scheduling scheme is known as "Systolic schedule".

In case of the generalized mapping, the above mentioned mapping procedures are case dependent and cannot be explicitly generalized. The scheduling and processor assignment will be non–linear in general. In the derivation of an SFG, a block of nodes is mapped on to a node irrespective of the arc directions connecting different nodes. This often helps in deriving SFGs for irregular DGs with global connections (e.g.,FFT).

3.3.2.3. Algebraic approach for SFG projection:

The above procedure can be described in terms of algebraic transformations. Given a DG of dimension n, a projection vector $\vec{d}$, and a permissible linear systolic schedule $\vec{s}$, then an SFG can be derived based on the following mappings.

3.3.2.3.1. Node mapping:

This assigns node activities to processors. The index set of the nodes of the SFG are represented by the mapping

$$P{:}R \longrightarrow I^{n-1}$$

where R is the index set of nodes of DG and I^{n-1} is the cartesian product of (n–1) integers. The mapping of a computation C in the DG on to a node n in the SFG is found by

$$n = P^T C$$

where the processor basis P, denoted by an n×(n–1) matrix is *orthogonal* to $\vec{d}$. Mathematically,

$$P^T \vec{d} = 0$$

3.3.2.3.2. Arc mapping:

This mapping maps the arcs of the DG to the edge of the SFG. The set of edges $\vec{e}$ into each node of the SFG and the number of delays $d(\vec{e})$ on every edge are derived from the set of dependent edges $\vec{b}$ at each point in the DG by

$$\begin{bmatrix} D(\vec{e}) \\ \vec{e} \end{bmatrix} = \begin{bmatrix} \vec{s}^T \\ P^T \end{bmatrix} \begin{bmatrix} \vec{b} \end{bmatrix}$$

3.3.2.3.3. I/O mapping:

The SFG node position, n, and time t(c), of an input of the DG computation is derived as

$$\begin{bmatrix} t(c) \\ n \end{bmatrix} = \begin{bmatrix} \vec{s}^T \\ P^T \end{bmatrix} \begin{bmatrix} c \end{bmatrix}$$

3.3.3. Partitioning:

In general, the above mapping procedure leads to an array, whose number of PEs is a function of the size of the problem. In practical systems, the size of the problem rarely matches the array size. As goes the Murphy's law, "***No matter what the special purpose system hardware available, there is always a problem too large for it***".

The partitioning problem is basically mapping computations of a larger problem to an array processors of smaller size. In general, the mapping scheme (both node assignment and scheduling) will be much more complicated than the regular projection methods discussed earlier. The following factors should be taken into account while partitioning a problem.

(1) Minimum overall computation time.

(2) Minimum control overhead.

(3) Balanced trade—off between external communication and local memory.

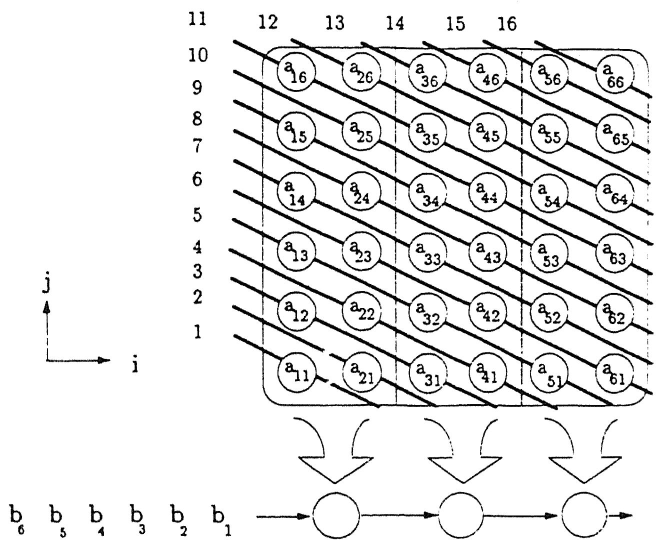
For a systematic mapping from the DG on to a systolic array, the DG is partitioned into many blocks, each consisting of a cluster of nodes. There are two methods of partitioning the DG.

(1) Locally sequential globally parallel (LSGP)

(2) Locally parallel globally sequential (LPGS)

3.3.3.1. LSGP scheme:

In LSGP scheme, one block is mapped to one PE. Thus, the number of blocks is equal to the number of PEs in the array. Each PE sequentially executes the nodes of the corresponding block. In order to store the node data in the block, local memory within each PE is needed. As long as local memory is large enough for computation under consideration, the LSGP scheme is quite appealing.



11
12
13
14
15
16
10
9
8
7
6
5
4
3
2
1
j
i
a_{16} a_{26} a_{36} a_{46} a_{56} a_{66}
a_{15} a_{25} a_{35} a_{45} a_{55} a_{65}
a_{14} a_{24} a_{34} a_{44} a_{54} a_{64}
a_{13} a_{23} a_{33} a_{43} a_{53} a_{63}
a_{12} a_{22} a_{32} a_{42} a_{52} a_{62}
a_{11} a_{21} a_{31} a_{41} a_{51} a_{61}
b_6 b_5 b_4 b_3 b_2 b_1

Fig. 3.5. LSGP Partitioning Of A Graph.

3.3.3.2. LPGS scheme:

In the scheme, the block size is chosen to match the array size; i.e.,one block can be mapped to the whole array at a time. All nodes within one block are processed concurrently (locally parallel). One block after another block of node data are loaded into the array and processed in a sequential manner (globally sequential). In this scheme, the local memory can be kept constant independent of the size of the problem. All intermediate results are stored in buffers outside the array. However, there is going to be an increase in the communication overhead by using this scheme, which may some times severely reduce the efficiency of parallelization.

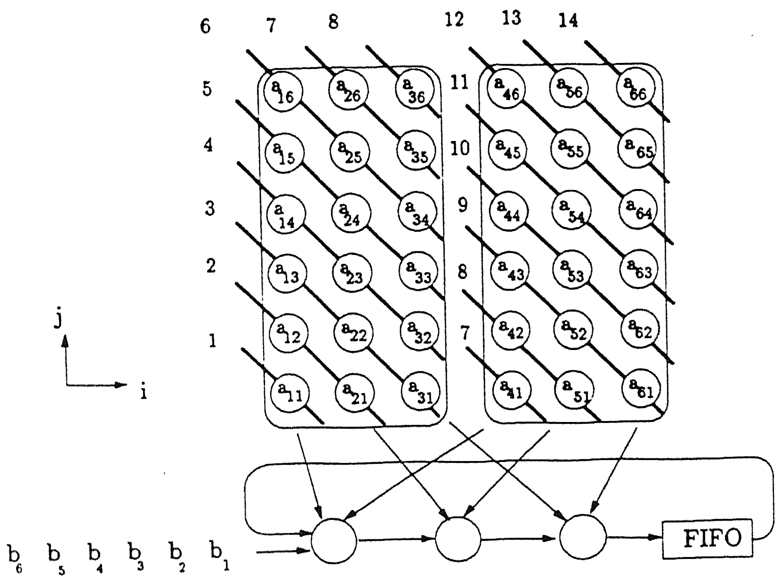


Fig. 3.6. LPGS Partitioning Of A Graph.

CHAPTER 4

DESIGN OF SYSTOLIC ALGORITHMS FOR TRANSPUTERS

4.1.Introduction:

In Chapter 3, we ~~have~~ described a general methodology for designing Systolic algorithms. Now we take up the design of a few specific algorithms.

We concentrate on two popular mapping methodologies for mapping algorithms onto processor arrays.

(1) ***Canonical mapping*** methodology for mapping homogeneous DGs

(2) ***Generalized mapping*** methodology for mapping heterogeneous DGs.

Before designing any parallel algorithm for Transputers, the following point should always be kept in mind. Because the Transputers are networked using point–to–point serial links and function as a message passing system, their data transfer capabilities are limited. In fact, it is observed that in most problems, the Transputer's arithmetic processing is so much faster than data communication along its links that any attempt to increase interprocessor communication as would be inevitable when implementing ***fine grain parallelism*** will severely reduce the efficiency of a parallel algorithm. In essence, the Transputers are not suitable for fine grain parallelism. In order to derive efficient Systolic algorithms, we will have to adopt LSGP partitioning scheme. In effect, we

(a) make the grain size of computation as large as possible.

(b) reduce the interprocessor communication overhead as much as possible.

A Systolic cell is supposed to do simple operations. But because of the above mentioned reason, our Transputer cell operation cannot be simple and is bound to be much more complex. Hence, here in this context, we have to compromise on the definition of a Systolic cell.

4.2. Mapping Algorithms To Array Processors:

4.2.1. The Canonical mapping methodology:

There are a large number of algorithms that can be expressed in terms of a very regular and localized DG. By exploiting this regularity, the array processor design for such algorithms may be greatly simplified. Briefly speaking, in the canonical mapping methodology, the regularity of the DGs is exploited and linear projection and scheduling schemes are adopted, so that simple and regular array structure may be achieved.

We take up three algorithms for parallelization and see how we can implement them using a fixed number of systolic cells.

4.2.1.1. The Matrix Multiplication:

If A and B are two N×N matrices then the their product, denoted by C, is given by C = AB.

The elements of C are:

$$C_{ij} = \sum_{k=0}^{N} a_{ik} b_{kj}$$

This equation implies that all the multiplications involved can be carried out at the same time since they do not have any dependencies between each other. To get maximum parallelism, we need to propagate input data to N^3 multipliers; two input links for each multiplier. That means at least $2N^3$ communication links are necessary.

This motivates the adoption of some new variables, so that the matrix multiplication algorithm may be written in a more useful recursive form:

$$c_{ij}^{k} = c_{ij}^{k-1} + a_{ik} b_{kj}$$

where k is the recursive index.

Alternatively, it can be written as a sequential code, as follows:

```
FOR $i$ from 1 to N
FOR $j$ from 1 to N
FOR $k$ from 1 to N
$c(i,j,k) = c(i,j,k-1) + a(i,k)b(k,j)$
```

This algorithm is expressed in global form; i.e., $a(i,k)$ is broadcast to all index points that have the same i–index and k–index. Similarly, $b(k,j)$ is broadcast to all index points that have the same k–index and j–index. Therefore, global communication is implicitly implied by the code.

We can convert this representation into a localized one as:

```
FOR $i$ from 1 to N
FOR $j$ from 1 to N
FOR $k$ from 1 to N
$a(i,j,k) = a(i,j-1,k)$
$b(i,j,k) = b(i-1,j,k)$
$c(i,j,k) = c(i,j,k-1) + a(i,j,k)b(i,j,k)$
```

The DG corresponding to this algorithm is shown in fig 4.1. The functional operations executed in each node are also shown in the figure.

The SFG for the above algorithm is shown in fig. 4.2. The projection direction $\vec{d}^T$ is along [0 1 0], and a delay denotes an advance in the j–index.

Choosing the *Processor basis* such that it is orthogonal to the projection direction,

$$P = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

and the scheduling vector $\vec{s} = \vec{d}$,

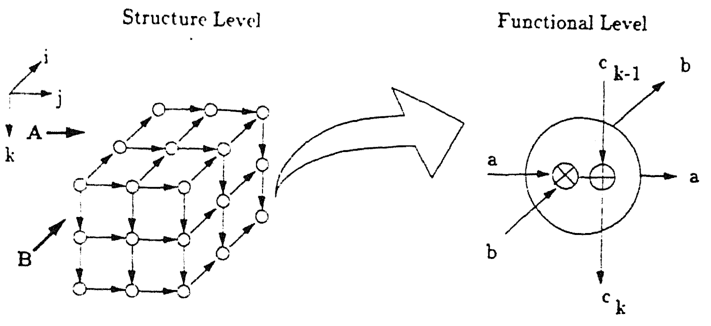


Fig. 4.1. DG For Matrix Multiplication.

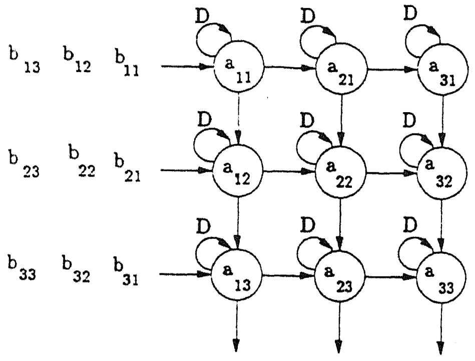

D
b_{13} b_{12} b_{11}
a_{11} a_{21} a_{31}
b_{23} b_{22} b_{21}
a_{12} a_{22} a_{32}
b_{33} b_{32} b_{31}
a_{13} a_{23} a_{33}

Fig. 4.2. SFG (Projection along [0 1 0]) For Matrix Multiplication.

Node mapping:

$$P^T\vec{d}=\begin{bmatrix}1\,0\,0\\ \\0\,0\,1\end{bmatrix}\begin{bmatrix}i\\j\\k\end{bmatrix}=\begin{bmatrix}i\\ \\k\end{bmatrix}$$

Arc mapping:

$$\begin{bmatrix}D(\vec{e})\\ (\vec{e})\end{bmatrix}=\begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}=\begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}$$

I/O mapping:

$$\begin{bmatrix}t(c)\\ n\end{bmatrix}=\begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}\begin{bmatrix}1&i&i\\j&1&j\\k&k&N\end{bmatrix}=\begin{bmatrix}j&1&j\\1&i&i\\k&k&N\end{bmatrix}$$

Now, as we can see from the diagram, we require N^2 nodes for this problem. We can apply the procedure once again by mapping the SFG in the k–direction. This is shown in fig. 4.3. Thus the algorithm can be implemented on N nodes. In effect, an N x N matrix is divided in terms of its row vectors and each vector is stored in one Systolic cell. The column vectors of the other matrix are passed systolically through the linear array and the product calculation is carried out in each of the cells separately, independent of one another.Each Systolic cell will be storing one row of the resultant matrix at the end of the computation. This LSGP type of partitioning results in larger grain size and reduced communication overhead and hence a better algorithm.

When the number N exceeds the number of Systolic cells (processors) available, then more than one column is stored in each processors. Thus for multiplying N×N matrices using k cells (assume N is divisible by k for the simplicity of explanation), we store (N/k) rows of one of the matrices in each cell and pass the elements of the other matrix column wise systolically. The partial computations are carried out in the processors and the partial results are stored in the respective cells at the end of computation (See fig 4.4.).

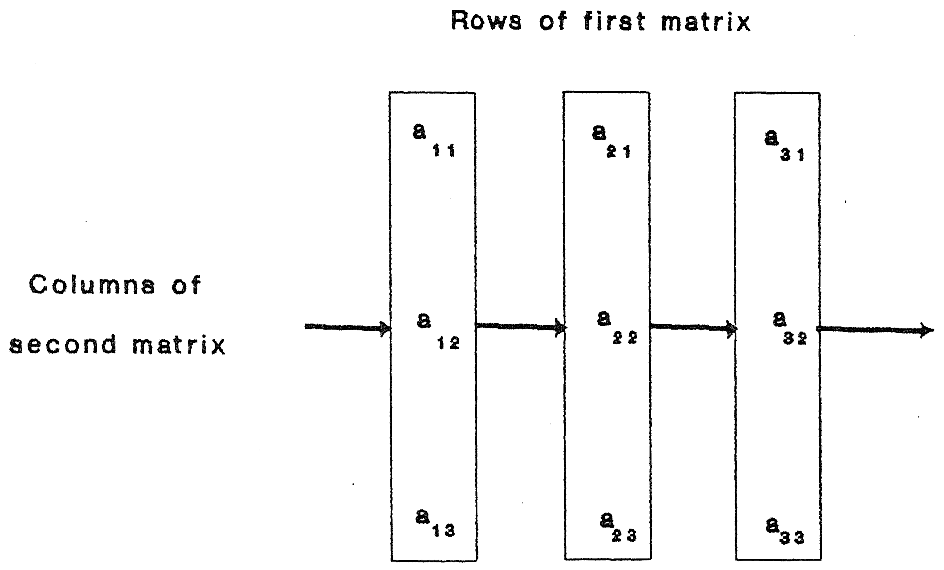


Fig. 4.3. Increasing Grain Size By Further Mapping Matrix Multiplication

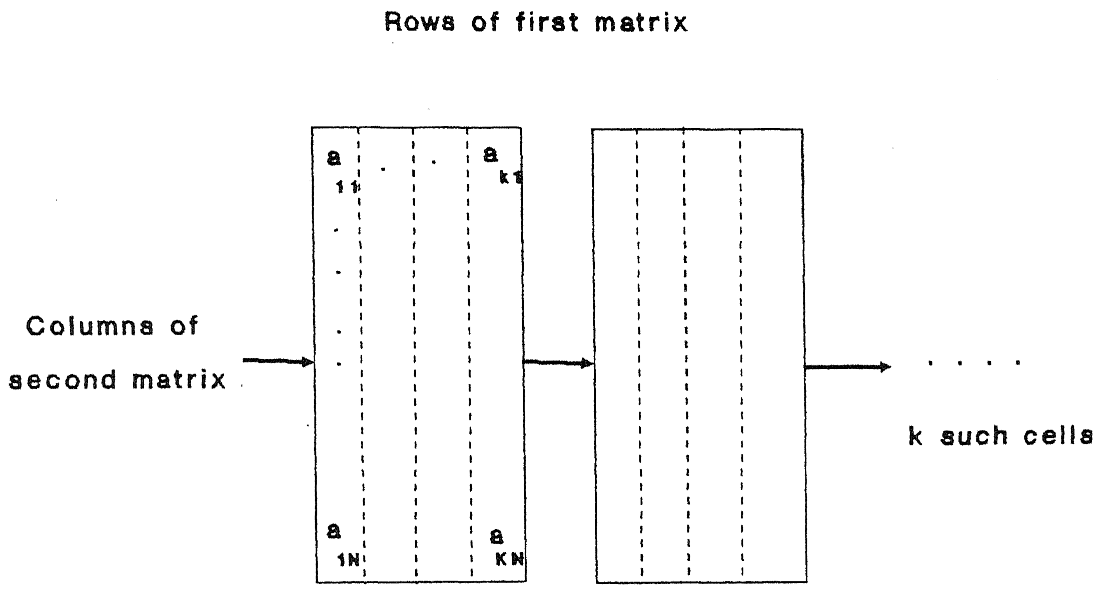


Fig. 4.4. Mapping To Smaller Size Array

The pseudo algorithm:

```
(1) Receive (N/k) rows of the first matrix.
(2) For i = 1 to N
    • Receive one column of the other matrix.
    • carry out the multiplications so that first k elements in the ith column
      are computed.
    • Send the column to the processor next to it.
(3) Send out the k rows of the resultant matrix.
```

4.2.1.2. The Convolution:

The problem of convolution is defined as follows: Given two sequences $u(j)$ and $w(j)$, $j = 0, 1, \ldots, N-1$, the convolution of the two sequences is

$$y(j) = \sum_{k=0}^{j} u(k)\, w(j-k)$$

or

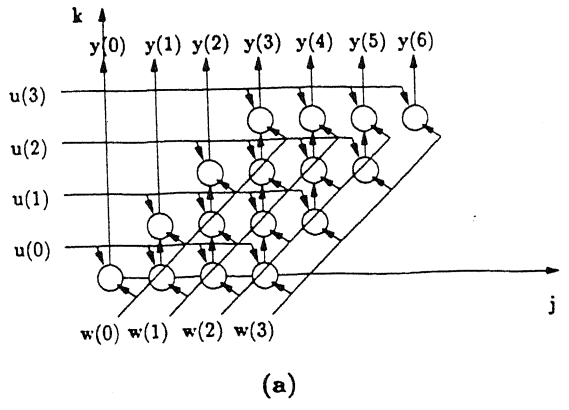
$$y_j = \sum_{k=0}^{j} u_j w_{j-k}$$

where $j = 0,1,\ldots,2N-2$.

The derivation of a DG for convolution is similar to matrix vector multiplication case. The first step of deriving the recursive equation is to introduce a recursive variable y_j^k. Then the convolution equation can be rewritten in terms of a recursive form:

$$y_j^k = y_j^{k-1} + u_k w_{j-k}$$

for $j = 0,1,\ldots,N-1$, and $k = 0,1,\ldots,j$ and for $j = N,N+1,\ldots,2N-2$, and $k = j-N+1, j-N+2,\ldots,N-1$.



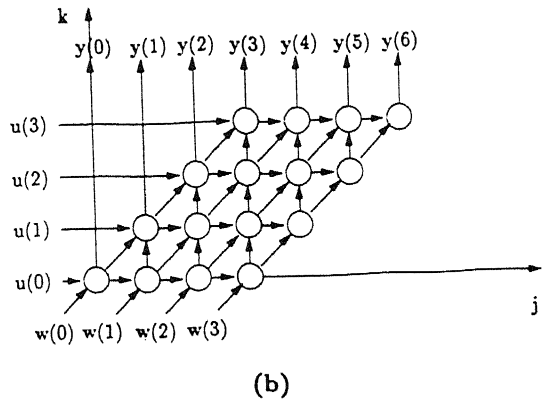


Fig. 4.5(a). DG For Convolution: (a) global;(b) localized.

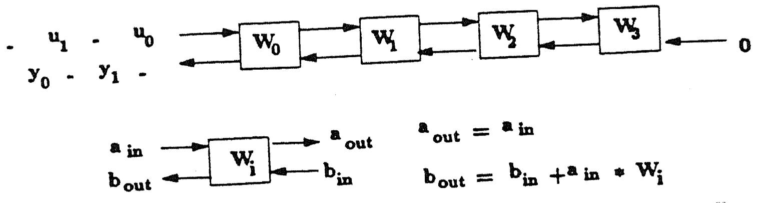


Fig. 4.5(b). SFG For Convolution.

The equation is an expression with global data dependencies and is hence to be localized. By replacing the broadcast contours by local arcs, the localized DG and hence the locally recursive algorithm is as follows (fig 4.5):

$$y^k_j = y^{k-1}_j + u^k_j\, w^k_j, \qquad y^0_j = 0$$
$$u^k_j = u^k_{j-1}, \qquad u^k_0 = u_k$$
$$w^k_j = w^{k-1}_{j-1}, \qquad w^0_{j-k} = w_{j-k}$$

for j = 0,1,...,N−1, and k = 0,1,...,j and for j = N, N+1,...,2N−2, and k = j−N+1, j−N+2,..., N−1.

We choose the projection vector $\vec{d}^T$ as [1 1], the scheduling vector $\vec{s}^T$ as [0 1] and the processor basis P as P = [1 −1] such that $P^T\vec{d} = [0]$.

Node mapping:

$$P^T\vec{c} = [1\ \ -1]\begin{bmatrix} i \\ j \end{bmatrix} = [i-j]$$

Arc mapping:

$$\begin{bmatrix} D(\vec{e}) \\ \vec{e} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} j \\ i-j \end{bmatrix}$$

I/O mapping:

$$\begin{bmatrix} t(c) \\ n \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} i & i \\ i & i \end{bmatrix} = \begin{bmatrix} i & i \\ 0 & i \end{bmatrix}$$

The above projection results in a SFG as shown in fig. 4.6. This is an N cell algorithm and for implementing this on fixed number of processors (say k), we assign the operation of n/k cells to one processor. This is eventually LSGP partitioning. The mapping of computation of (n/k) cells to one processors should be optimized. In case of Transputers, this optimization can be done at the source code level,so that the overhead due to process scheduling, increased interprocessor communication will be minimum after partitioning.

The pseudo algorithm:

For the ith cell,

```
(1)Load (n/k) points of one of the sequences to be convolved starting at $W_{i*k}$.
(2)For i = 1 to n
    • Receive the other sequence points one by one from the previous cell.
    • Compute the partial sum of the output point.
    • Receive the partial sum from the next cell.
    • Sum the received ansd calculated values.
    • Send the partial results back to the previous cell.
(3)For i=N+1 to 2N−1
    • Compute the partial result.
    • Receive the partial result from the next cell.
    • Send the summed partial result to the previous cell.
```

4.2.1.3 Sorting:

Sorting is one of the most frequently used procedures in data processing and scientific computation. The problem of sorting can be stated as follows:

Given a sequence $\{x(i)\}$, the problem is to obtain a new sequence $\{m(i)\}$, consisting of the elements of $\{x(i)\}$, rearranged into a sorted order. For example, the sequence $m(i)$ can be considered to be sorted in a decreasing order if $m(i) \geq m(k)$ when $i<k$.

Sorting algorithms can be easily formulated into a locally recursive form. The single assignment program of a selection sorting can be written as:

```
For $i$ from 1 to N
For $j$ from 1 to N
        $m(i+1,j) \longleftarrow max[x(i,j),\ m(i,j)]$
        $x(i,j+1) \longleftarrow min[x(i,j),\ m(i,j)]$
```

Here the $x(i,1)$ is initialized to the original unsorted sequence $x(i)$, and $m(i,j)$, is set to minus infinity, and the sorted output sequence $m(j)$ is equal to $\{m(N,j),\ j = 1,2,\ldots,N\}$ which is in decreasing order. The DG corresponding to the single assignment program is shown in fig 4.6. By inspection, the DG is clearly shift–invariant since the dependency structure do not vary from node to node.

We choose the following algebraic transformation:

$\vec{d}^{T} = \vec{s}^{T} = [1\ \ 0],\ P^{T} = [0\ 1]$

Node mapping:

$$[0\ \ 1]\begin{bmatrix} i \\ j \end{bmatrix} = j$$

Arc mapping:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

I/O mapping:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} i & N \\ 1 & j \end{bmatrix} = \begin{bmatrix} i & N \\ 1 & j \end{bmatrix}$$

The resulting SFG is a linear array and is shown in Fig 4.7. which corresponds to the ***insertion sorter***. The input data is sequentially read at the first node; and the results stay in each individual node. The unsorted sequence is fed to the first node and after exactly n cycles of operations, the list will be completely sorted.

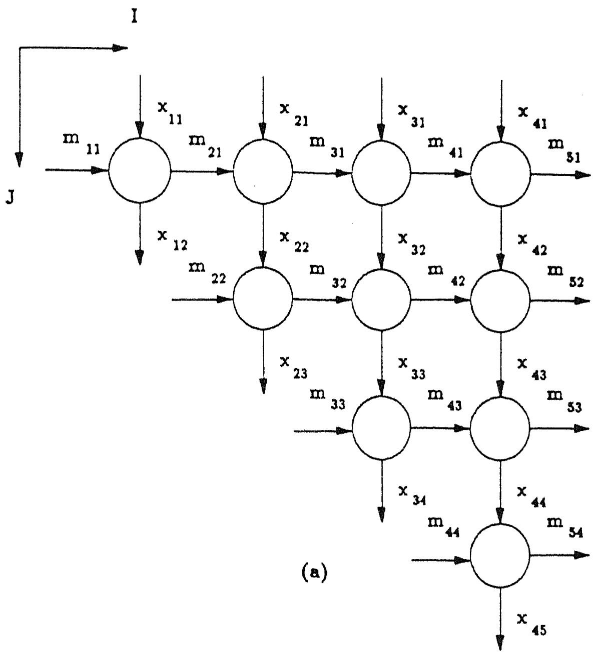


Fig. 4.6. DG For Sorting.

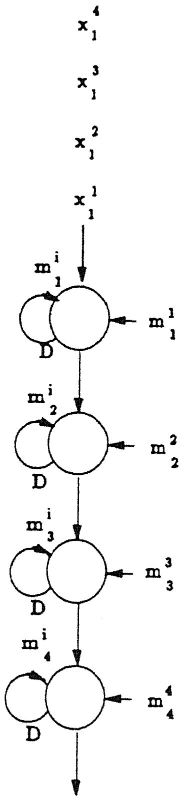


Fig. 4.7.

SFG Array For

Insertion Sorting.

The pseudo algorithm for the n–node array:

Receive an element and assume it is the maximum value.

For i=1 to N–1

- Receive an element
- compare it with the stored value. If the received value is greater, then swap the values.
- Send the smallest of the two numbers down the array.

For implementing this N–node problem on k Transputers, we use a simple technique of assigning the computation of (N/k) nodes to a single node i.e., LSGP partitioning of the problem. The resulting algorithm is optimized for implementation on a single node with minimum overheads resulted due to partitioning.

The pseudo algorithm for the k–processor array:

In each node,

(1)Receive the first (n/k) values one by one and sort them, each time finding the location of the value in the current list and inserting it in the location.

(2)While there is still input to sort,

- Receive a value.
- Go on comparing the value with the values in the already sorted list of size (N/k) and find the location where it is to be inserted in the list.
- Insert the new value in the list and send the minimum value in the list down the processor array.

4.2.2. The Generalized mapping methodology:

The class of homogeneous DGs already covers a broad range of algorithms. Nevertheless, further generalization is possible and desirable to many other important algorithms that are not totally regular, but still have certain degree of regularity. This semi regularity often proves to be useful for an efficient mapping methodology. The new approach will allow an extended DG classification to be dealt with and to have options on linear or nonlinear assignment/schedule.

4.2.2.1. The Fast Fourier Transform:

The problem of computing an N–point discrete fourier transform (DFT) is as follows: Given $x_0, x_1, \ldots, x_{n-1}$, compute $y_0, y_1, \ldots, y_{n-1}$, defined by

$$y_i = x_0 \omega^{i(n-1)} + x_1 \omega^{i(n-2)} + \ldots + x_{n-1}$$

where ω is a primitive Nth root of unity.

Assume N is a power of 2. The well–known fast fourier transform (FFT) method solves an n–point DFT problem in $O(n\log_2 N)$ operations, while the straightforward method requires $O(N^2)$ operations. The FFT involves $\log_2 N$ stages of N/2 butterfly operations and data shuffling between any two consecutive stages. The so called constant geometry version of the FFT algorithm allows the same data shuffling to be used for all the stages. This is depicted in fig 4.8. with n=16. In the figure, the butterfly operations are represented by circles, and number h by an edge indicates that the result associated with the edge must be multiplied by ω^h.

This diagram is nothing but a DG representation of the algorithm in which all the data dependencies are shown in the single assignment form. Though the DG is very regular, any attempt to use linear projection leads to an irregular SFG and hence the assignment should be non–linear.

The above algorithm can be very efficiently implemented on $\log_2 N$ Systolic cells. In the array, all the butterfly operations in the ith stage are carried out by cell i, and results

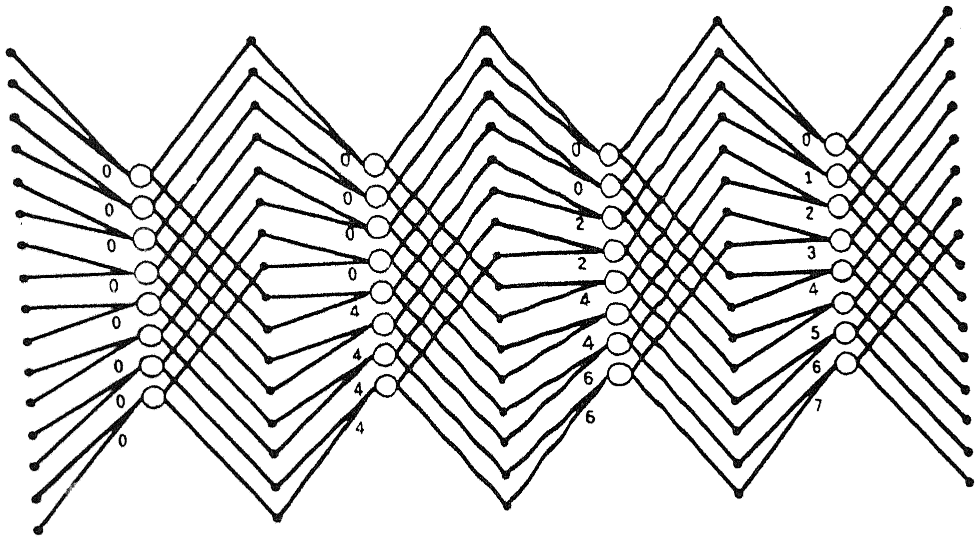


Fig. 4.8. Constant Geometry Version Of FFT Algorithm.

are sent to the next cell. While cell i works with the ith stage of an FFT, the $(i+1)$th cell will be working with the $(i+1)$th stage of another FFT. In practical applications, there are often a large number of FFTs to be processed and there are FFT problems continuously generated. Thus it is possible that a new FFT problem can enter the first cell, as soon as the cell becomes free. In this way, all the cells of the Systolic array can be kept busy all the time.

We now describe how butterfly operations are executed by each cell. A butterfly operation,

$$(a_r+ja_i)\pm(b_r+jb_i).(w_r+jw_i)=[a_r\pm(b_r.w_r-b_i.w_i)] \\ +j[a_i\pm(b_r.w_i+b_i.w_r)]$$

involves four real multiplications and six real additions. The twiddle factors required for the cell operation can be generated within the cell itself, thereby avoiding the need to pass those values also to the cells.

In case we have less number of cells, we can map the computation of more than one cell onto a processor. This is precisely the LSGP partitioning. The pipelined FFTs are produced from the array one after the other continuously.

The pseudo algorithm:

```
For the ith cell,
        Compute the twiddle factors required for the ith stage
        while there still data to process,
            •   Receive the data from the left end.
            •   Compute the ith stage FFT.
            •   Send the partial results to the next stage.
```

1.2.2.2. The Fast Walsh Transform:

The problem of computing n–point Walsh transform is as follows: Given $x_0, x_1, \ldots, x_{n-1}$, compute $y_0, y_1, \ldots, y_{n-1}$, defined by

$$y_i = \sum_{j=0}^{n-1} (-1)^{i \otimes j} x_j$$

where $i \otimes j$ is the Dyadic sum of the two n–bit integers i,j given by

$i \otimes j = \sum_{k=0}^{n-1} i_k\, j_k$, where i_k, j_k are the kth bits when i and j are represented as n bit numbers.

The basic functions of the Walsh transform are Walsh functions which are binary valued with {1,–1}. Thus generation and implementation of walsh transform is simple. furthermore, the only basic operations involved are additions and subtractions, which means the cost of computation is low.

The WHT can be formulated as a matrix–vector multiplication similar to the DFT. The Hadamard matrix is a square array of +1s and –1s, whose rows and columns are orthogonal. Similar to Fourier transform, the Hadamard transform has a fast algorithm called fast Hadamard transform.

The structure of the fast Hadamard algorithm is very much similar to that of a constant geometry. It has $\log_2 N$ stages. The computation in all the stages is identical. The data shuffling between adjacent stages is also identical. This is depicted in fig.4.9. The circles represent an addition and a subtraction operation.

The procedure we use here for calculating the Walsh transform is similar to that we used for FFT. We require $\log_2 N$ cells for calculating the transform. The projection is again a non–linear projection and one stage is mapped to one cell. When the ith cell does the ith stage calculation of a WHT, the $(i+1)$th stage will be busy with the $(i+1)$th stage of another transform. Thus it is possible to process a series of WHTs in a pipelined fashion.

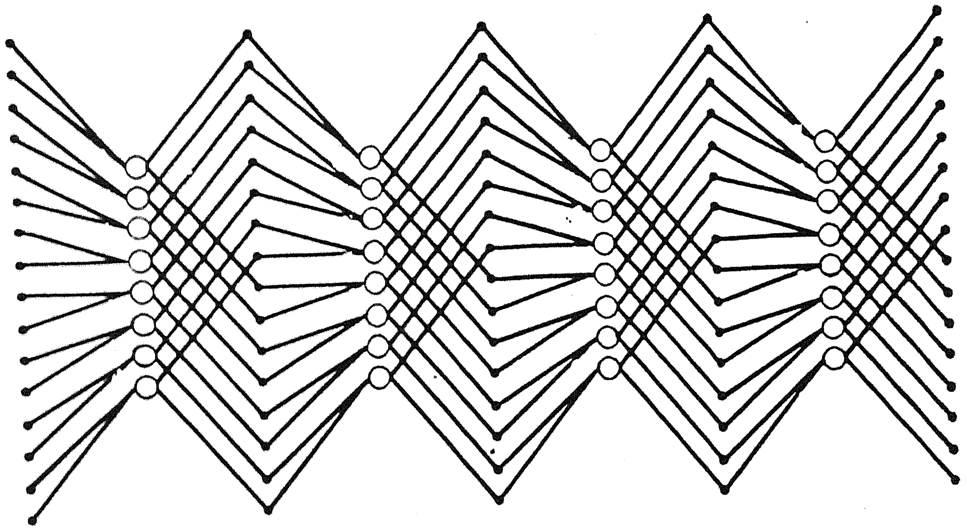


Fig. 4.9. The Structure Of Fast Walsh Transform.

The cell operation is given by

$$b_i = a_{2i} + a_{2i+1}$$

$$b_{i+k} = a_{2i} - a_{2i+1} \qquad \text{Where } k = N/2$$

Needless to specify that this is LSGP partitioning. Fig. 4.10. gives the n cell systolic structure. If we have less number of cells, we have to map the computation of more than one stage onto a processor which is a practical consideration.

The pseudo algorithm:

```
For the ith cell,
        While there is still data to process,
            •  Receive the sequence from the earlier cell.
            •  Compute the ith stage of the input sequence.
            •  Send the partial result down the array stream.
```

CHAPTER 5

TRANSPUTER AND OCCAM

5.1.The Transputer [3]:

5.1.1. Introduction:

"Transputer" can be used as a basic element in multiprocessor systems. The chip, being very versatile, has received considerable attention and popularity among concurrent system designers recently.

The word *Transputer* may be interpreted as a contraction of the words *Tranceiver* and *computer*. The interpretation suggests that the Transputer consists of a communication system and a computational element.

There are a number of interesting features of the Transputers which distinguish the Transputer from any other processor. Of all, the integration of the design and the advantage of having many features built into a single chip are the important aspects, which result in the enhancement of processor speed, low chip count, and simplicity of the system design.

A Transputer can be used in a single processor system or in networks to build high performance concurrent systems. A typical member of transputer product family is a single chip containing processor, memory and point–to–point communication links. A network of Transputers can be easily constructed using these links. As a microcomputer, the Transputer is unusual in its ability to communicate with other Transputers. A variety of different configurations can be built by hard–wiring Transputers together, with no separate switching and forwarding network, limited only by the number of links provided on each Transputer. The current Transputer systems have four to six links. Four links are enough to allow enormous range of useful configurations.

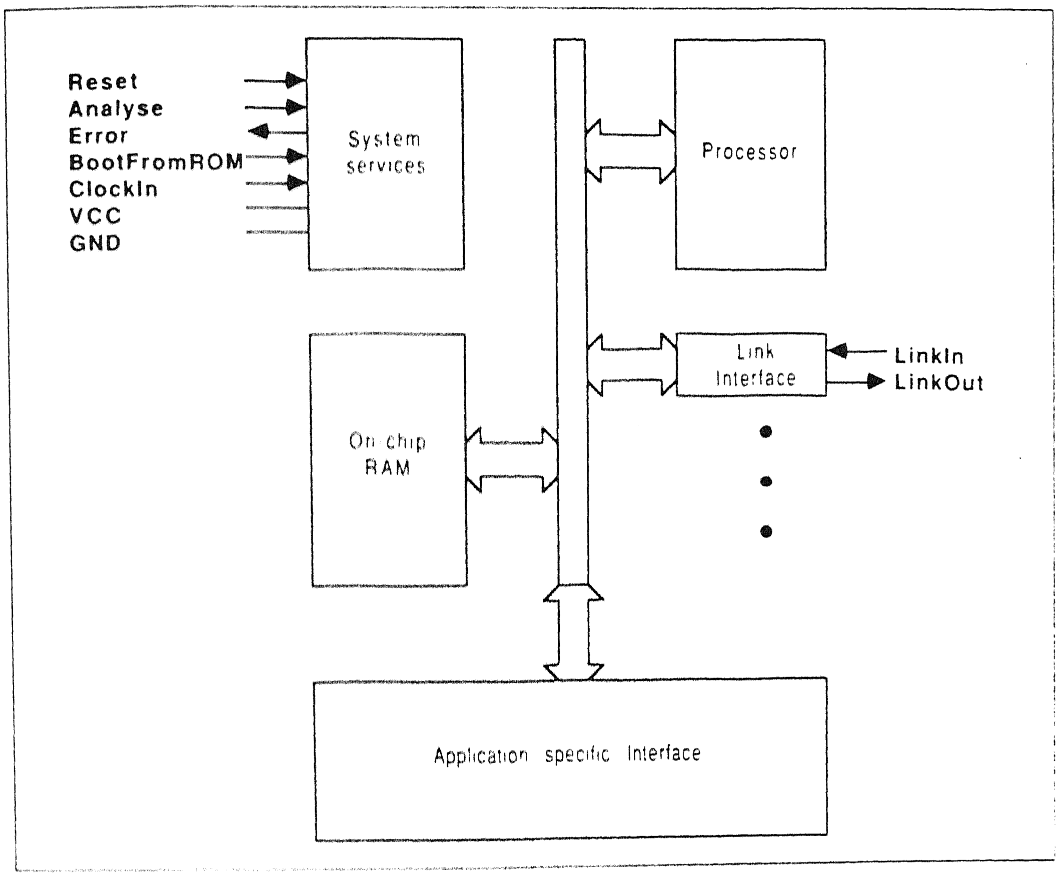


Fig. 5.1. Transputer Architecture.

Scalability is a major advantage of transputer–based systems: it is much easier to enhance a system by adding further processors to it than would be the case with other microprocessors. *Compatibility* – the ease of replacing one transputer model with another without major design changes in a system is also valuable. This extends to mixing models of transputer within one system.

5.1.2. Architectural details:

Transputers are often described as RISC processors. The computational instructions follow RISC principles closely, and they attain the benefits claimed for RISC architecture. However, they also have a small number of important non–RISC instructions concerned with scheduling and message passing.

The Transputer is unusual in its ability to execute many software processes at the same time. A program can be run on a single Transputer, in which case the concurrency of the processes will be simulated by hardware with no software intervention. Provided the communication between the subprocesses is not too complicated, the same program can also be distributed over several processors, in which case the component processes will be run in real concurrency. Just as in a single processor, interprocess message passing and the necessary synchronization (between directly connected Transputers) are achieved in hardware, and no operating system is needed.

The inbuilt features of the Transputer make up a long list. As well as the instruction processor and small amount of on chip memory, it has a memory controller, DMA control for four independent fast links, a microcoded multitasking kernel, and an elapsed–time clock (fig. 5.1). The best known Transputers are the T212, T414 and T800. The T212 is a 16–bit processor, the other two are 32–bit processors. The T800 has a full IEEE floating–point processor on chip.

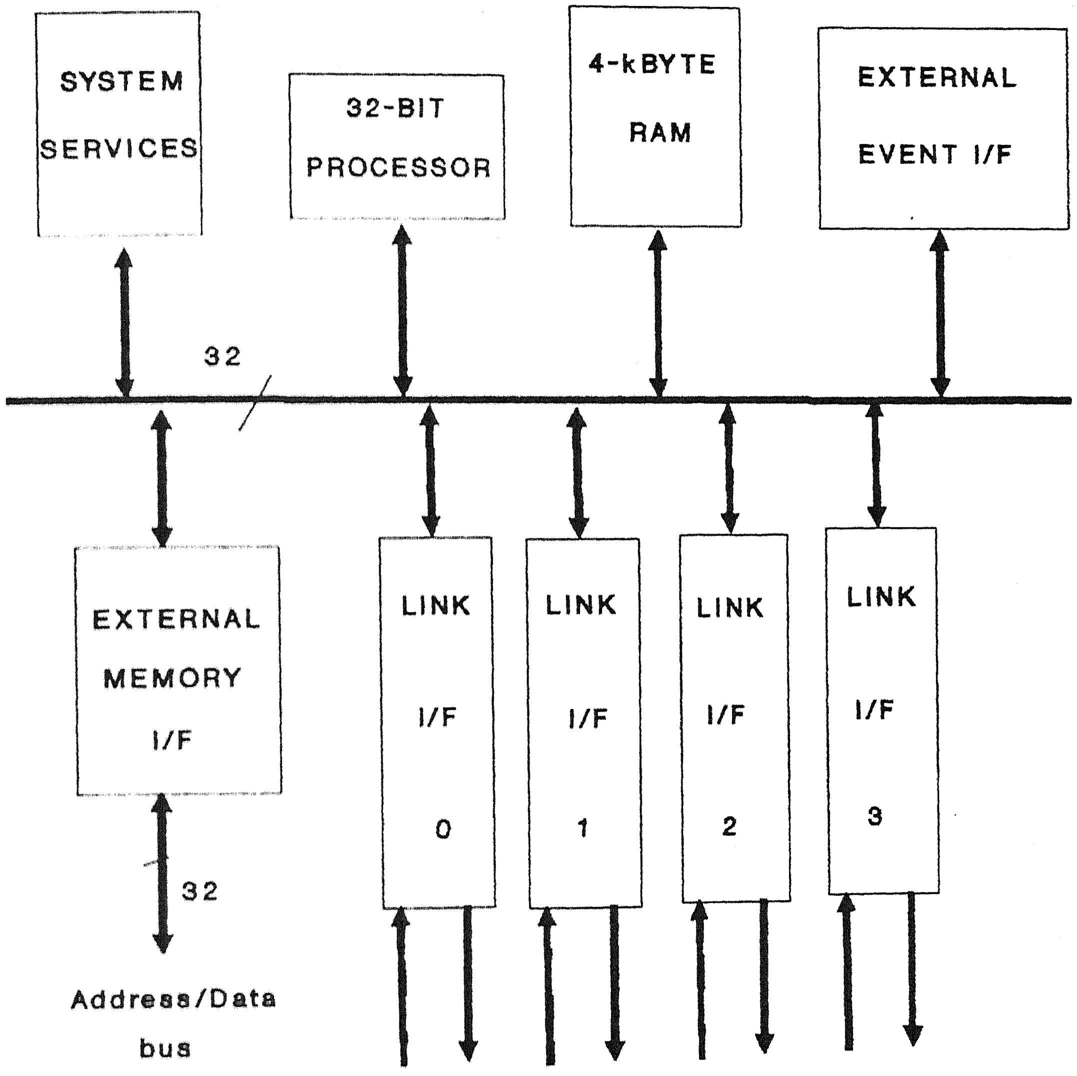


Fig. 5.2. INMOS T800 Transputer

5.1.2.1. Instruction processor:

The processor portion of a Transputer is a traditional microprocessor. Fig shows the 32–bit version of the Transputer. The processor normally obtains its instructions and data from the internal 4K RAM .Data and instructions can also be obtained from the links. The processor provides 32–bit addressing. memory is addressed byte–wise and stored in 4–byte units. Software sees no difference between on–chip and external memory except in speed.

The design of Transputer processor exploits the availability of fast on–chip memory by having only a small number of registers; the CPU contains six registers which are used in the execution of a sequential process. The small number of registers, together with the simplicity of the instruction set enables the processor to have relatively simple (and fast) data–paths and control logic.

The registers are:

- The workspace pointer which points to an area of store where local variables are kept.
- The instruction pointer which points to the next instruction to be executed.
- The operand register which is used in the formation of instruction operands.
- A short, internal three register stack for expression evaluation (integer and address arithmetic).

The instructions refer to the stack implicitly. The use of a stack removes the need for instructions to respecify the location of their operands. The stack need not be saved when rescheduling occurs. Statistics gathered from a large number of programs show that three registers provide an effective balance between code compactness and implementation complexity.

5.1.2.2. **Instruction set:**

Transputer instruction set is designed for simple and efficient compilation. All the instructions have the same format and chosen to give a compact representation of the operations most frequently occurring in programs. The instruction size of the Transputer is only 8 bits for most instructions. There are prefix instructions which allow the operand to be extended to any length. Measurements show that about 70% of the executed instructions are encoded in a single byte. Short instructions improve the effectiveness of the instruction prefetch, which in turn improves processor performance. There is an extra word of prefetch buffer, so the processor rarely has to wait for an instruction fetch before proceeding. Since the buffer is short, there is little time penalty when a jump instruction causes the buffer contents to be discarded. Also, the instruction set is independent of processor word length, allowing the same microcode to be used for Transputers with different word lengths.

5.1.2.3. Memory controller:

The memory controller can drive external dynamic RAM with no additional circuitry. Together with the controller, the processor can address a linear address space of 4–Gbytes. The 32–bit wide memory interface uses multiplexed data and address lines and provides a data rate of up to 4 bytes every 100 nanoseconds (40 Mbytes/sec) for a 30 MHz device. The configurable memory controller provides all timing, control and DRAM refresh signals for a wide variety of mixed memory systems.

5.1.2.4. Process scheduler:

The processor provides efficient support for concurrency and communication. It has a microcoded scheduler which enables any number of concurrent processes to be executed together, sharing the processor time. This removes the need for a software kernel. The processor does not need to support the dynamic allocation of the storage as the compiler is able to perform the allocation of space to the concurrent processes.

A process starts, performs a number of actions, and then terminates. Typically a process is a sequence of instructions. A transputer can run several processes in parallel (concurrently). Processes may be assigned either high or low priority. At any time, a concurrent process may be

active	–	being executed
	–	on a list waiting to be executed
inactive	–	ready to input
	–	ready to output
	–	waiting until a specified time

The scheduler operates in such a way that inactive processes do not consume any processor time. The active processes waiting to be executed are held on a list. This is a linked list of process workspaces, implemented using two registers, one of which points to the first process on the list, the other to the last (fig. 5.3).

The IMS T800 supports two levels of priority. Priority 1 (low priority) processes are executed whenever there are no active priority 0 (high priority) processes.

High priority processes are expected to execute for a short time. If one or more high priority processes are able to proceed, then one is selected and run until it has to wait for communication, a timer input, or until it completes processing.

If no process at high priority is able to proceed, but one or more processes at low priority are able to proceed, then one is selected.

Each process runs until it has completed its action, but is descheduled whilst waiting for communication from another process or Transputer, or for a time delay to complete. In order for several processes to operate in parallel, a low priority process is only permitted to run for a maximum of two time slices before it is forcibly descheduled at the next descheduling point. The time slice period is 5120 cycles of the external 5 MHz clock, giving ticks approximately 1ms apart. If there are n low priority processes, then the maximum latency from the time at which a low priority process becomes active to the time when it starts processing is $2n-2$ time slice periods. It is then be able to be executed for

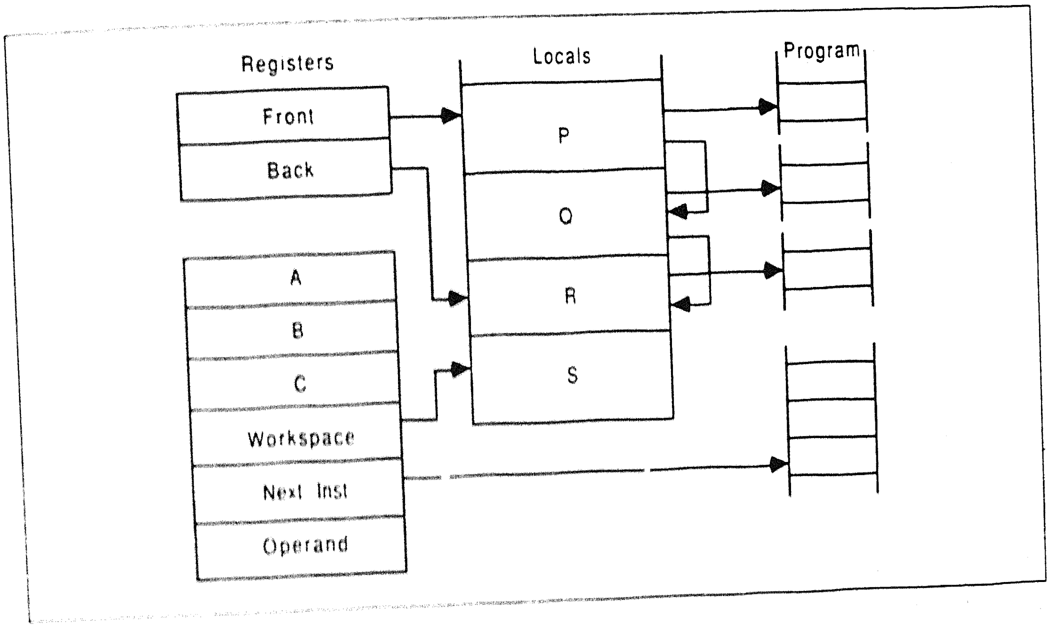


Fig. 5.3. Linked Process List.

A process can only be descheduled on certain instructions, known as descheduling points. As a result, an expression evaluation can be guaranteed to execute without the process being time sliced part way through.

Whenever a process is unable to proceed, its instruction pointer is saved in the processor workspace and the next processor is taken from the list. Process scheduling pointers are updated by instructions which cause scheduling operations, and should not be altered directly. Actual process switch times are less than 1 μs, as little state needs to be saved and it is not necessary to save the evaluation stack on rescheduling.

The processor provides a number of special operations to support the processor model, including start process and end process. When a main process executes a parallel construct, start process instructions are used to create the necessary additional concurrent processes. A start process instruction creates a new process by adding a new workspace to the end of the scheduling list, enabling the new concurrent process to be executed together with the ones already being executed. When a process is made active it is always added to the end of the list, and thus cannot pre—empt processes already on the same list.

The correct termination of a parallel construct is assured by the use of the end process instruction. This uses a workspace location as a counter of the parallel construct components which have still to terminate. The counter is initialized to the number of components before the processes are started. Each component ends with an end process instruction which decrements and tests the counter. For all but the last component, the counter is non zero and the component is descheduled. For the last component, the counter is zero and the main process continues.

5.1.2.5. Communications:

Communication between the processes is achieved by means of channels. The communication is point–to–point, synchronized and unbuffered. As a result, a channel needs no process queue, no message queue and no message buffer.

A channel between two processes executing on the same Transputer is implemented by a single word in memory; a channel between processes executing on different Transputers is implemented by point–to–point links. The processor provides a number of operations to support message passing, the most important being

input message *output message*

The input message and output message instructions use the address of the channel to determine whether the channel is internal or external. This means that the same instruction sequence can be used for both for hard and soft channels, allowing a process to be written and compiled without the knowledge of where its channels are connected.

The communication takes place when both the inputting and outputting processes are ready. Consequently, the process which first becomes ready must wait until the second one is also ready.

A process performs an input or output by loading the evaluation stack with a pointer to a message, the address of a channel, and a count of the number of bytes to be transferred, and then executing an input message or an output message instruction.

5.1.2.6. Communication links:

A link between two transputers is implemented by connecting a link interface on one Transputer to a link interface on the other Transputer by two one–directional signal wires, along which data is transmitted serially. The two wires provide two channels, one in each direction. This requires a simple protocol to multiplex data and control information. Messages are transmitted as a sequence of bytes, each of which are to be acknowledged before the next is transmitted. A byte of data is transmitted as a start bit followed by a one bit followed by eight bits of data followed by a stop bit. An acknowledgement is transmitted as a start bit followed by a stop bit. An acknowledgement indicates that a process was able to receive the data byte and that it is able to buffer another byte.

The protocol permits an acknowledgement to be generated as soon as the receiver has identified a data packet. In this way the acknowledgement can be received by the transmitter before all of the data packets have been transmitted and the transmitter can transmit the next data packet immediately. The IMS T414 transputer does not implement this overlapping and achieves a data rate of 0.8 Mbytes per second using a link to transfer in one direction. However, by implementing sufficient overlapping and including sufficient buffering in the link hardware, the IMS T800 more than doubles this data rate to .8 Mbytes per second in one direction, and achieves 2.4 Mbytes per second when the link carries data in both directions.

5.1.2.7. The Timer:

The Transputers have two timer clocks which 'tick' periodically. The timer provide accurate process timing, allowing processes to be descheduled until a specific time.

In the IMS T800 Transputer, one timer is accessible to only high priority processes and is incremented every microsecond, cycling completely in approximately 4295 milliseconds. The other is accessible only to a low priority process and is incremented every 64 microseconds, giving exactly 15625 ticks in one second. It has a full period of approximately 76 hours.

5.1.2.8. The Floating point processor:

The IMS T800 has a full IEEE floating point processor on chip. The FPU operates concurrently with CPU. This means that it is possible to do address calculation in the CPU whilst the FPU performs the floating point calculation. This can lead to significant performance improvements in real applications which access arrays heavily. Performance depends on many things, including clock and memory speeds — for the 20 MHz T800 these figures are of the order of 10 RISC MIPS and 1 MFLOPS (these are not upper bounds).

As can be noticed from above, all the links can be active at the same time as well as the processor. Thus the Transputer can support nine truly concurrent activities. (one link can transfer data in both directions, of course, memory accesses have to be interleaved). On a T800, the floating point processor operate in parallel with the instruction processor, which gives a tenth level of concurrency at the hardware level (but both the processors are controlled by a single instruction stream).

5.2. Occam [5]:

Occam's Razor – ***Entities are not to be multiplied beyond necessity*** was the philosophy of the original implementation of Occam.

Occam is the native language for Transputers. The design of Occam has been heavily influenced by the work on *Communicating Sequential Processes* (CSP), which gives a mathematical frame work for specifying the behaviour of parallel processes. Occam is based on the CSP model of computation, but with features chosen to ensure efficiency of implementation. In this model, an application is decomposed into a collection of communicating processes, and the processes communicate by passing messages.

Occam model is based on the idea of ***process***. The software building block is a ***process***. A system is designed in terms of an interconnected set of processes. Each ***process*** cab be regarded as an independent unit of design. Its internal design is hidden and is completely specified by the message it sends and receives. Internally, each process can be designed as a set of communicating processes. The system design is therefore hierarchically structured. Occam processes do not share any variables, nor semaphores. Occam does not require (nor support) shared memory. Messages pass from exactly one process to the other. There are no multiple senders or receivers, no broadcasting, and no uncertainty about where a message came from or where it is going. Messages are unbuffered, so sending and receiving a message involves momentary synchronization between the two participating processes. Messages are sent through static channels, as if through a circuit switched (rather than packet switched) network.

To gain the most benefit from the Transputer architecture, the whole system can be programmed in Occam. This provides all the advantages of a high level language, the maximum program efficiency and the ability to use the special features of the Transputer. The Occam model of concurrency is applicable equally to processes running on separate processors and to processes running within a single processor. Since the processor can be controlled only by one instruction stream, it is evident that ***the processes in one processor cannot be truly concurrent***. However, the processes can be multiprogrammed, just as on a mainframe, so that the effect of concurrency is reproduced (apart from speed). This 'simulated concurrency' as distinct from 'real concurrency' is known as '***gratutious concurrency***'. Within a Transputer, this multiprogramming is handled by hardware with no need for any operating system.

Occam provides a framework for designing concurrent systems using Transputers just in the same way that boolean algebra provides a framework for designing electronic systems from logic gates. The system designer's task is eased because of the architectural relationship between Occam and Transputer. A program running in a Transputer is formally equivalent to an Occam process, and so a network of Transputers can be described directly as an Occam program.

Occam, when compiled for execution on a Transputer, is ideal for embedded multiprocessor systems. Where it is required to exploit concurrency, but still to use standard languages, Occam can be used as a harness to link modules written in selected languages. Performance approaching that of assembled machine code can be achieved, by both matching of an architecture to a specific language, and the use of static memory allocation avoiding run–time memory range checking.

5.3. Programming Transputers and experience [6]:

The following are the points that concern the programmer of a concurrent system.

5.3.1. Topology:

The pattern in which the processors are connected together is known as ***topology*** or the configuration. The idea of 'configuration' depends on the assumption that processors are connected permanently, or at least for the life of a whole program, an assumption which is broadly true for current Transputer systems.

It is the responsibility of the designer of the overall system to decide how to configure the processors within it. Designing a topology for a large system can be difficult. It can sometimes be guided by an obvious mapping of a problem (or a solution) onto separate processors, but the designer is not always so fortunate and most often analyse several difficult possibilities using what help is obtainable from concurrency theory, graph theory, queuing theory etc. There is a strong tendency to fall back on standard, straightforward and well understood topologies even through they may be far from optimal.

5.3.2. Placement:

Describing systems in terms of Occam processes allows algorithmic issues to be separated from the question of what hardware is going to perform those activities. This is a useful abstraction. One would like to be able to express an algorithm in the form of a program which is independent of hardware, so that it could be subsequently be performed using many different networks of processors. Each implementation would need a specification of how many processors were needed, how they were to be connected, and which processes were to be installed on which processors, but the specification ought not to need any change in the program. The specification is called ***placement.***

In practice, Transputer programs are not completely independent of their ***placement.*** Unless it is carefully designed, a program will only run on one particular network of Transputers, or on a small number of similar networks. To run on other networks, the program itself will have to be changed. Sometimes the changes will be minimal, but othér cases may need extensive modifications. Programmers therefore have to make a conscious effort to write programs which can easily be run on different configurations.

5.3.3. Non–determinacy:

Sequential programmers are used to the idea of a ***bug***. There are solid bugs and intermittent bugs. Intermittent bugs are data dependent; a program run on the same inputs will work every time or it fails every time. Concurrent programming has a third type of bug: the bug that depends on the relative timing of concurrent processes. These are very often not repeatable, even if the program is rerun on the same data.

The problem arises because all the processors are allowed to run at their own speed. ***There is no attempt to constrain the processors into a lock step***. Thus the order of events can change from one test run to another.

It is obviously important at the design phase not to assume an exact ordering of the events. One would also expect it to be impossibly hard to test and debug Occam software, yet programmers commonly do succeed. The key to success is to reduce the need for testing to an absolute minimum, and to understand exactly what Occam defines as the effect of a program and what it leaves undefined. Occam is as precise as any other language about what individual processes do, but it cannot specify the relative timing of concurrent processes (otherwise the whole point of concurrency would be lost).

It is the programmer's responsibility to ensure that when a program terminates it has completed the required function, regardless of the order in which things happened between starting and termination. Since Occam does not guarantee determinacy, it has been designed to express and handle non–determinacy very simply, flexibly, and elegantly.

5.3.4. Deadlock:

A classic problem in concurrent systems is ***deadlock***. This is an affliction whereby one part of a program is waiting for another to do something; the other is waiting for the first to do something else, and since both are waiting, neither can do what other expects. There may, of course, be a set of processes involved, rather than a pair.

All Transputer deadlocks are essentially the same in that they involve a closed chain of processes, each trying to communicate with another, but with no pair of them willing to participating in any one communication. There is an enormous variety of ways which may lead to deadlock, and that makes it hard to avoid. It is also difficult to analyse and hard to cure after it is found to occur in a program. Formal methods used are writing only simple code, supported by intuition and back–of–envelope sketches.

5.3.5. Concurrent algorithm design considerations:

Most of the concurrent algorithms are only loosely related to their sequential equivalents. The conversion of a sequential algorithm into concurrent algorithm, often called '***parallelization***' is too ill defined and difficult to be automated and is often attempted by hand. The following points should be given due consideration while developing the concurrent algorithms.

5.3.5.1. Granularity:

The term '***granularity***', which is jargon much used among concurrent programmers, refers to the size of the task that are distributed as concurrent processes. In a matrix multiplication, all the elements of the result can, in principle, be evaluated concurrently, because none of the calculations depends on the result of any other. That would be ***fine grain concurrency***. A more ***coarse grain*** division of labour could evaluate one quarter of the result, working sequentially on one processor, while evaluating the other three quarters on three other processors.

Intuitively, one would expect fine–grained concurrency to deliver results faster but to use more processors. With the Transputers, a *finer–grain* division of work requires more information to be passed between processors. ***Reducing the grain size below some optimum value for a particular problem can incur message passing costs which grossly outweigh the expected benefits.***

Granularity is a particular problem in the conversion of existing sequential code into concurrent methods. It is relatively easy to recognize fine–grain potential parallelism, e.g.several assignments whose order is immaterial, or a simple loop. It is much harder to identify the larger units of a program which might safely be run concurrently.

5.3.5.2. Performance measure and efficiency:

One measure of the quality of a concurrent system is its efficiency in using the processors. If a single processor solution takes n seconds, it is desirable to achieve a solution with m processors in n/m seconds. Complete efficiency is never attainable, but the user tries to get near to it. Efficiency must fall off as more processors are added.

Occasionally it does happen that an n–processor mechanism solves a problem in less than m/n seconds, which require m seconds on a single processor. This appears to show more than 100% efficiency. There are three possible explanations. One is that the speed is data dependent: if several sets of data are tried, all of which require m seconds on one processor, then some may need much less than m/n seconds on one processors, but others will need more than m/n seconds, so that the average performance is enhanced by a factor less than n. Characteristically, these problems have extremely variable solutions even on a single processor; i.e., the time needed to process one set of data is is not obviously related to the times needed by other sets of data of apparently similar difficulty.

Another possible explanation is that, in recasting an algorithm for concurrent running, a better method than the original program may have been developed. For example, the sequential program might be doing more page swapping than the concurrent algorithm which result in the ***superlinear efficiency***.

It seems, then, that it sometimes possible for practical Transputer users to realize gains in performance larger than the number of processors used, but there is always some excuse for denying that this shows more than 100% efficiency. This suggests that a more sophisticated measure of efficiency is needed. In reality, the user should never expect to benefit from the super efficiency, for that occurs only rarely.

5.3.5.3. Balancing communication and processing:

Some times, the concurrent algorithm may show very poor efficiency. the reasons could be

- The computation has not been divided equally — one of the processor still has to do far more than 1/n th of the work.
- The processes are independent; they have to share or communicate information via links, and some of them spend a lot of time waiting for others, during which time they cannot continue with the computation.
- Even if time is not wasted in waiting for messages, there may lot of time spent in passing the messages.

These three possibilities are quite distinct. The first one has to be solved by '***load balancing***' — attempting to equalize the computation load on each processor. The second can be described as '***spurious concurrency***'. It can occur by accident, and it is not easy to predict, analyse or detect. Even when it is known that it is happening, it is not easily cured. The third reason for inefficiency is a difficult optimization problem between computation and communication times.

5.3.5.4. Software development [4]:

Transputer software is mostly developed under the '***Transputer development environment***' (TDS) supplied by INMOS.

TDS and related systems provide an integrated environment for editing, compiling and running the programs. They are centered on the '***folding editor***' which embodies an elegant and general way of representing and handling large amount of Occam text within a small screen. ***They lack some of the support tools that are common in other systems.***(*e.g.,in UNIX, diff, grep, multitasking, batch files, aliases, email, .login files.*)

The TDS is so much an '***integrated environment***' that its files are not easily handled in other systems, not even in systems such as MS–DOS which is acting as the host or file server for TDS. This makes it hard for the utility software on the host system to provide the facilities that are missing from the TDS.

Programmers would very much like to have further help in the peculiar difficulties of concurrent program development. The strongest demand at the moment, and the one which seems nearest to fulfillment, is for software tools for profiling, monitoring, and debugging.

5.3.5.5. Needs remaining unsatisfied:

There are several areas in which user's experience has established a requirement that no hardware or software has yet fulfilled. These users need a more comprehensive (and more familiar) software development environment, with tools for designing and constructing concurrent programs as well as more normal services for sequential programming, module management, etc. Extensive and usable monitoring and debugging facilities should be available, and more accessible formal methods, with software tools to support their use, would be welcome. The problem of file capacity, transfer rate and back–up have still to be solved.

On the hardware side, it is widely felt that the Transputer should use its on–chip memory as cache store, and that it should provide at least some support for memory management and protection. More links per Transputer would be welcome, as would automatic forwarding of messages between processors which are not directly connected.

Nevertheless the processing speed, the ease of multiprocessor interfacing and the availability of a naive programming language with attractive features make the Transputer a very good candidate for multiprocessing.

CHAPTER 6

IMPLEMENTATION OF SYSTOLIC ALGORITHMS ON TRANSPUTER NETWORKS

6.1. Introduction:

In Chapter 4 we have parallelized a few algorithms using standard systolic designs. Now, we will see how to implement them on a transputer network.

6.2. General System Organization of an Array Processor [1]:

The possible overall configuration of a general array processor system is depicted in Fig.6.1. It consists of the following major components:

- Host computer and/or array control unit (ALU).
- Interface unit.
- PE array(s).
- Interconnection network(s).

6.2.1. Host Computer:

Generally, the processor array is used as an attached processor to a general purpose host running an operating system. The processor array can be accessed by a procedure call on the host, or through an interactive, programmable command interpreter. The array system is in general *asymmetric* (master–slave). The host treats the arrays just as a special resource. The main extension to the conventional operating system is a driver for the arrays which must be able to treat the arrays both as a whole (to allocate tasks to arrays) and as a collection of processors (to load programs and data to individual PEs).

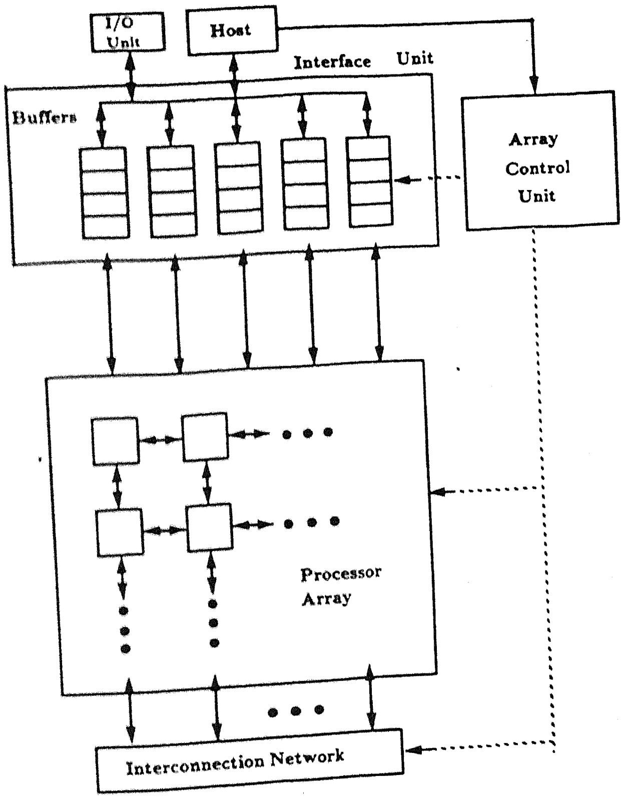


Fig. 6.1. Array Processor System.

The host computer is intended to provide system monitoring, data storage and management.It should determine the schedule program that controls all the system units, and generate global control codes and object codes of PEs. It can be a micro computer, workstation, minicomputer, main frame or a supercomputer. The selection depends on the desired application. It is important for the system designer to identify in advance a suitable host machine that can effectively support the array processor unit.

6.2.2. The Interface unit:

The interface unit is an interface between the host and the processor array. The interface unit, connected to the host via host bus, or DMA, has the function of down–loading, uploading, buffering array data and handling interrupts.

The major function of the interface unit is to support high bandwidth communication (accompanying high–speed processing) between the array and the host. By providing sufficient buffering, the unit must be able to balance the low bandwidth of the system I/O and the high bandwidth of the array processors.

6.2.3. PE arrays:

A PE array comprises of a number of processor elements with local memory. Most existing array processor systems (e.g.,systolic, wavefront, and SIMD arrays) emphasize both the fact that the PEs execute the same instruction, and that the PEs are interconnected in a regular and expandable manner.

An important factor in the design of the PE array is the local memory available to each PE as it gives maximum flexibility to a PE. By providing sufficient local memory (both on chip and off–chip), the PE can effectively utilize its temporary data storage thereby saving communication time and avoid tying up of the interface bus.

6.2.4. The Interconnection Network:

The interconnections within the PE array is provided by the interconnection network. These networks are generally large switching networks to provide flexibility of interconnections and are intended to provide global, local and high speed communications between the processing elements.

6.3. Hardware setup Of The Transputer Network System [7]:

A nine–node transputer network has been set up in the Image Processing Lab at IIT Kanpur (Fig 6.2). The setup contains two IMS B003 evaluation boards, each having four transputers and an IMS B004 transputer system development board having a single transputer. Of these nine transputers (INMOS, UK), five are T800s and four are T414s. The host is an IBM PC–AT (80386 based). One of the transputers (T800 type) functions as the interface unit, known as the ***Root node***, with 4 Mbytes of on board RAM. The host is connected only to the root node via a ***Link adapter*** (IMS C012). The root node performs the I/O between the processor array and the host. The PEs are the individual transputers each with 256 Kbytes of RAM on board. The transputer links are interconnected by a programmable switch known as the ***Link switch*** (IMS C004).

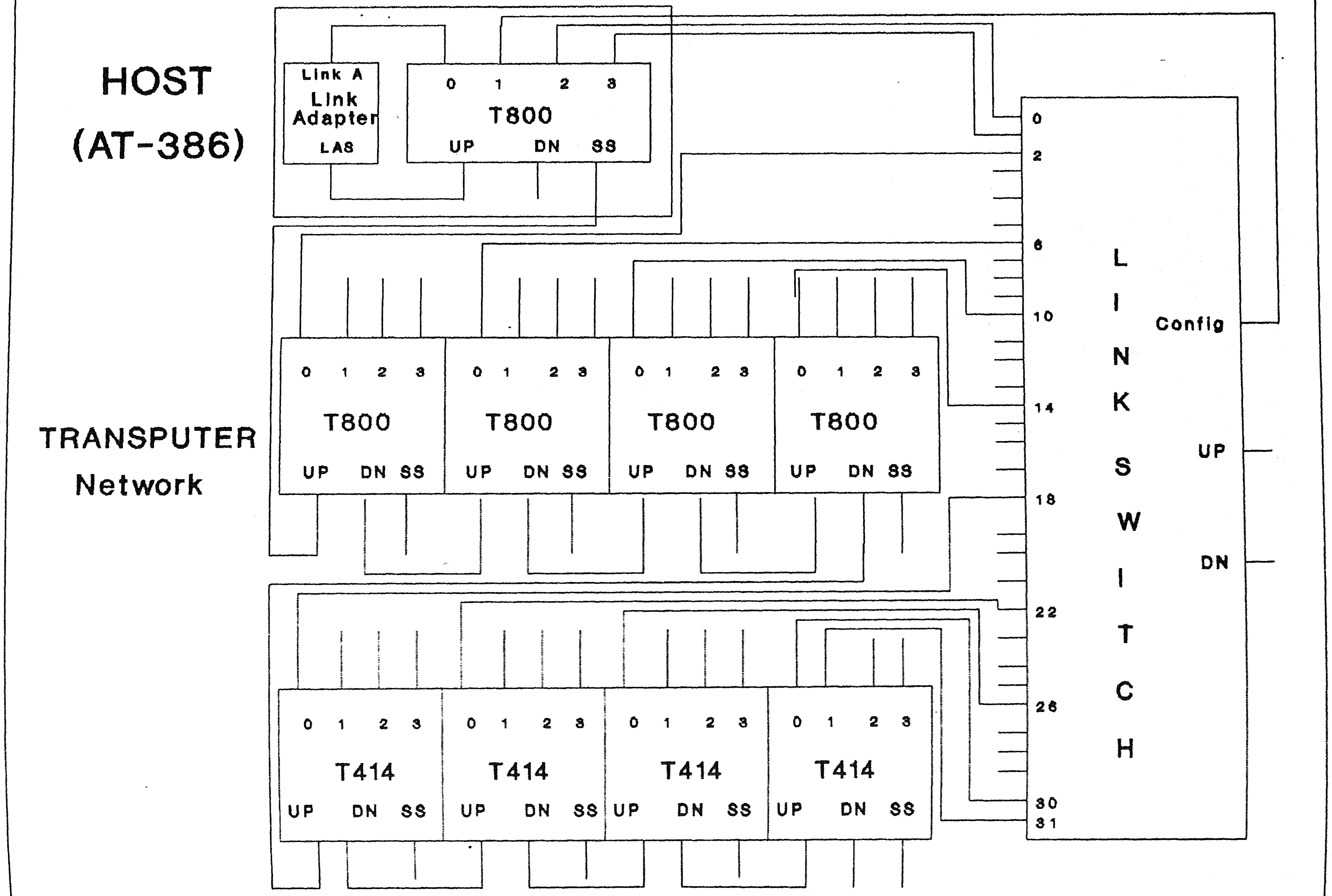
6.3.1. Link adapter:

The ***Link adapter*** connecting the root node and the host provides for full duplex transputer link communication by converting bi–directional serial link data into parallel data streams.

6.3.2. The host:

The host functions as a file server for the system as memory management and file system are not supported by the transputers. The server reads a DOS file to determine the network configuration (the configuration is specified at the compile time), the programs to

TIPS Hardware Setup



be loaded and determines the boot order. The host loads the network via the ***Root transputer*** by sending loading information to it. The root transputer will boot the transputers connected to it, and route loading information to them; these will in turn boot and load other transputers in the network, until the whole network has been booted and loaded. The topology of the actual physical network should match that described in the program configuration description, otherwise the loading will fail.

6.3.3. The interface unit:

The root transputer acts as an interface unit between the host and the network. All the communication between the host and the individual transputers takes place through the root transputer. Most of the times the root transputer will be used for executing the I/O routines required for the host file server. Unlike dedicated interfaces, this can also share the computation load with other transputers.

6.3.4. Booting of the transputer network:

A communication protocol exists between the host transputer and a target transputer network to direct the code to the desired place in each transputer. The communication consists of bootstrap packets, routing information, address information, code packets and execute items.

The bootstrap code for each transputer in the network is sent first. The bootstrap code is loaded in the lowest available address. The bootstrap loads the distributing loader at the first available address above itself. After all the transputers in the network are booted, the code of each of the procedures allocated to processors in the configuration description is exported to the network preceded by the necessary routing and loading information. Following this, the code which calls the procedures (the main body) generated by the configurer is sent to each processor in turn and then each processor is told to start executing the loaded program. (a detailed description of the loading mechanism is given in INMOS technical Note 34 '*Loading Transputer Networks*').

6.3.5. Network monitor:

INMOS boards provide system control functions to monitor and control the state of the transputer network. The system control connections on board are (daisy) chained together to allow the whole of the network to be controlled from the host. The control connection consists of three signals.

- Reset This is a signal from the host transputer to the network, which will reset all the transputers in the network, ready for loading.
- Analyze This is a signal from the host transputer to the network, which bring all of the transputers in the network to a controlled halt, so that their state can be examined.
- Error This is a signal from the network to the host transputer, indicating that one of the transputers in the network has set its error flag.

6.3.6. The Linkswitch:

The transputer network is interconnected using ***Link switch*** (IMS C004). It is a transparent programmable link switch designed to provide a full crossbar switch between 32 link inputs and 32 link outputs. It introduces on the average only a 1.75 bit time delay on the signal. Link switches can be cascaded to any depth without loss of signal integrity and can be used to construct configurable networks of arbitrary size. The switch is programmed via a separate link called the ***configuration link***.

In the present setup, LINK0 of the root transputer is connected to the host (via link adapter) and LINK1 is connected is used as the configuration link for the linkswitch. So, only LINK2 and LINK3 on the root transputer are free. Also, since the linkswitch supports only 32 link connections, two links of the last (9th) transputer cannot be used at present (only LINK0 and LINK1 can be used on the last transputer).

6.4.Implementation issues:

6.4.1 Systolic structure [8]:

The transputer network is intended to run as a linear one dimensional systolic array for our application (Fig.6.3). In general, the array takes inputs at the left most cell and produces the outputs at the rightmost cell, with data flowing in one direction. There are several advantages of having this simple interconnection scheme, besides the relative ease of algorithm design, implementation and use. Linear arrays require minimum possible I/O, in the sense that only the two end cells communicate with the outside world (interface unit) Thus an n cell processor can perform O(n) computation for each I/O operation, even if a constant number of computations are performed at each cell. This property is desirable in practice, because usually the I/O bandwidth is the major limiting factor for achieving high performance.

6.4.2. Array synchronization [9]:

Array synchronization is an important issue. It will now be argued that the transputer network, being a message passing system, can never be synchronous in the strict sense. ***The question is how the implementation of a systolic array on a message passing system like the transputers is justified.*** In transputer systems, we have no mechanism for explicitly synchronizing an entire array; i.e., making the data move from processor to processor in tune with a global reference clock, as is expected of a proper systolic array. A systolic cell would have to write (into its succeeding cell) and read (from its preceding cell) ***simultaneously*** at the start of every clock cycle. But for a transputer, the read and the write constitute two separate instructions that can be executed only one after the other (or on a time–sharing basis) and so are incapable of being actually executed truly in parallel

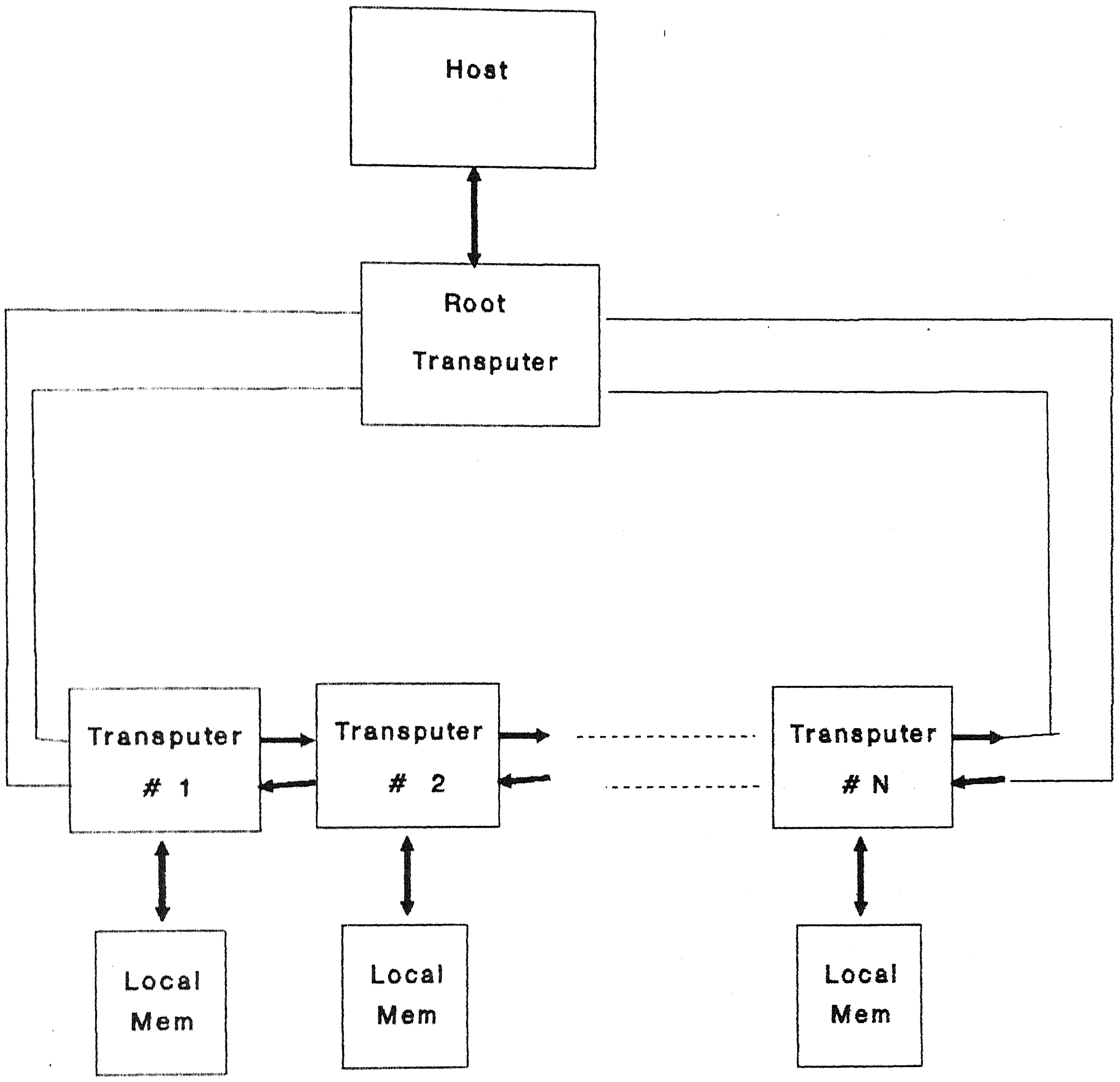


Fig. 6.3: Linear One Dimensional Systoliv Array
Of Transputers

within the single device. The '*gratutious*' nature of the parallelism adopted in a transputer that was described earlier suggests that if two processes are to be executed in parallel on a single processor, they are not actually executed in parallel, but only interleaved to ~~seem~~ be parallel to the user. For this reason, the point to be noted is that there is no way of making the transputer array genuinely synchronous. Some digression from the main stream of the discussion is needed to clarify this point.

Consider the primitive systolic cell model which is commonly used in descriptions of systolic array operation and synthesis techniques. It has a simple synchronous protocol and at each beat of the clock, it receives data from its previous cell, and outputs the results of its past computation to the next cell. Excepting the boundary nodes, all cells do the same computations. We have a point to make here that though all the cells start computing at the same clock cycle, the actual data for the ith cell in the array is sent only after a delay of i clock cycles from the start of the computation of the ith cell. This needs, (1) either (i–1) dummy data sets to be sent to the ith processor by the previous cells before it starts its actual computation or, (2) the ith cell to be made to wait till it gets its actual input operands by skewing the computation by i clock cycles. We may also note that this problem could arise not only during startup, but also when the processors execute instructions whose computation times are unequal on account of being data dependent; then, too, array synchronization could be lost.

It might appear that if the cells were synchronized explicitly, less hardware and software support would be needed to generate efficient code, even though the programming task would become more complicated. This is a misconception. Synchronization forces the time–evolution of a computation to become rigid and insensitive to variations in data that may alter the actually required time for computing the outputs in each cell. If, in a string of data, there are present elements of data that need less time to be computed than others, a synchronized system remains unable to exploit the advantage and continues to operate at the rate corresponding to the data taking the longest time to be computed. Moreover, the user has to manually arrange the clock rate to suit the time required for computing the data in hand. Synchronous computation models in which the user completely specifies all timing relationships between different cells are inadequate for high performance arrays. The reason is that they are hard to program and difficult to code efficiently. If the primitive systolic model were adopted as the machine model, the versatility of the CMU Warp machine (the prototype of systolic array processors) would have been severely impaired.

The problem is bypassed elegantly in Warp design by adopting asynchronous communication between the cells in the array. In Warp computer,

(a)Cells send and receive data to and from dedicated buffers: this can smoothen the communication delays between the cells.

(b)A cell is blocked when it tries to send data to a full queue or receive from an empty queue, till the queues get ready.

Thus the synchronization employed in the Warp computer is rather *forced* (distributed) synchronization. If we assent to term even this as a form of 'synchronization', then it is indeed true that transputers can support synchronized computing. Hence it all depends on how we interpret the word '***synchronized***' array.

6.4.3 Code development:

Software development is the major consideration. Occam is used for the program development under Transputer Development System (TDS) environment. The program development has two phases. In the first phase, the partitioned program is coded and debugged in occam and tested as a parallel program of n processes running on a single transputer using soft channels for communication between the parallel processes. The TDS system has extensive library support even for most of the low level applications which can be used for simplifying the programming task.

In the second phase, the same program is run on a processor network with the following changes made to the above program.

1. The individual processes are to be defined as procedures whose parameters should be only the hard channels and a processor identification number (sharing of variables between different processors is not posssible) and should be separately compiled separately. The global constants can be shared by using libraries.

2. A mapping of the occam channels to the transputer hard links should be given. This gives the network configuration information.

3. The separately compiled procedures are 'PLACED' on individual transputers and the program is compiled to generate a network code file.

For running the program, network linkswitch should be configured. This can be done by a software program which will transmit appropriate code to the configuration input link of the linkswitch.

The multi node program can be tested from the TDS environment. After successful run, it can be made a standalone program bootable by the external host by using the alien file server library routines available in the TDS environment.

Though Occam is more suitable in terms of generating efficient code for transputers than any other language, it, being very primitive, cannot support a wide range of applications. For example, it cannot support a variety of data structures because it does not have pointer data structures. All the memory allocation is static and done at the compile time, and no dynamic allocation of memory is supported.

6.4.4. Code optimization [3]:

The following guide lines should be kept in mind for code optimization while writing programs in Occam.

- 1. Use constants rather than variables wherever possible.
- 2. Channel slices are very much more efficient than single width channels. (This is especially significant when implementing channel buffers.)
- 3. Retyping of multidimensional arrays as single dimensional arrays results in a significant increase in efficiency if array index can be determined without a multiplication.
- 4. Use INTs rather than BYTEs.
- 5. For short operands PROD is very much faster than MUL.
- 6. ALT command should contain as few guard commands as possible.

CHAPTER 7
PERFORMANCE EVALUATION

7.1. Introduction:

The performance of a parallel algorithm is an important consideration. The Speedup factor of a parallel algorithm is defined as the ratio between the time needed for the most efficient sequential algorithm to perform a computation and the time needed to perform a computation on a machine incorporating pipelining and/or parallelism [11].

The time complexity of a parallel algorithm is a function $f(n)$ that is the maximum, over all inputs of size n, of the time elapsed from when the first processor begins execution of the algorithm until the last processor terminates algorithm execution. The cost of a parallel algorithm is defined as its complexity times p, the number of processors.

7.2. Speedup factor for Systolic arrays:

As a special case, the systolic array algorithms are expected to exhibit linear rate pipelinability; i.e., it should achieve an O(M) speedup, in terms of processing rate, where M is the number of processing elements.

The times of execution of the algorithms on an array of 8, 4, 2, 1, transputers respectively are as shown in Table 7.1. The speedup factor is given in Table 7.2. The execution times of the algorithms on IBM PC AT are also shown in Table 7.1. In most cases, the array is exhibiting near linear speedup rate. Thus we can obtain an effectiv systolic array with hardware complexity and processing time.

7.3. Performance comparison with Warp:

CMU's Warp computer is a prototype Systolic processor. Hence it is reasonable compare the performance of any systolic array processor with that of Warp.

We discuss architectural decisions made in Warp by contrasting them with the decisions made in bit–serial processor arrays, such as transputer. We know that the transputers are used extensively for computer vision and image processing. However, their architecture is significantly different as compared to Warp.

Bit–serial processor arrays implement a *data parallel* programming model in which different processors process different elements of the data sets. In systolic arrays, the processors individually manipulate the data, whereas in Warp data parallel programming models are implemented through the use of input and output partitioning. In addition to this, the high interprocessor bandwidth of the systolic array allows efficient implementation of pipelining, in which the algorithm is partitioned and not the data.

Bit serial processor arrays suffer from a serious bottleneck in I/O with the external world because of the difficulty of feeding a large amount of data through a single simple processor. The high bandwidth of the Warp makes it possible for all processors to see all data while achieving useful processing time.

The Warp cell is more powerful compared to the Transputer as far as implementing pipelined architectures because of the massively functional parallelism employed in the cell. The most common operations like reading the buffer for a particular execution unit and writing into the buffer of the next cell are encoded in the same instruction by employing long Horizontally microcoded instructions. Transputer does not have the necessary functional parallelism support as required for systolic arrays. These are the reasons for the poor performance of the transputer array compared with the Warp array.

In essence, the bit–serial processor arrays excel in processing data partitioning models such as encountered in edge detection at lowest level in an Image Understanding Systems. However, specialized hardware must be used to eliminate a severe I/O bottleneck when used for algorithm partition model.

Warp gives good performance in a wide range of algorithms, especially those including complex global computations on moderately sized data sets.

Sl.No	Algorithm	Time of Execution(ms)					
		On Transputer					On PC
		1 Node	2 Node	4 Node	8 Node	Sequential	Sequential
1.	Matrix Multiplication (64×64)	2194	1046	602	385	2044	4286
2.	Walsh Transform (256 point) 100 transforms at a time	1494	838	509	350	1493	1868
3.	Convolution (1024 points)	6843	4818	2925	1576	5957	6318
4.	FFT (256 point) 100 FFTs at a time	47409 (T414)	2776 (T800)	1244 (T800)	6103 (T414)	47481 (T414) 4718 (T800)	11648
5.	Sorting (1024 point) Best case Worst case	800 1667	527 1406	413 864	286 477	3 1525	<55 2637

7.4. Suggestions for future work:

A more general class of complex algorithms like LU decomposition and solving of partial differential equations may be attempted. Also, the future generation transputers are expected to have word–wide data transfer capabilities. They are also expected to have more links. Hence by employing more number of transputers, a performance close to that of Warp may be achieved.

Sl.No	Algorithm	Speedup factor					
		On Transputer			On IBM PC–AT		
		2 Node	4 Node	8 Node	2 Node	4 Node	8 Node
1.	Matrix Multiplication	1.95	3.39	5.3	4.1	7.12	11.13
2.	Walsh Transform	1.78	2.93	4.26	2.23	3.67	5.34
3.	Convolution	1.24	2.04	3.78	1.31	2.16	4.00
4.	FFT	1.7	3.79	7.78	4.20	9.36	1.91
5.	Sorting						
	Best case	—	—	—	—	—	—
	Worst case	1.08	1.765	3.2	1.875	3.05	5.3

Sl.No	Task (all images are 512×512)	Time(ms)	Speedup over Vax 11/780
1.	Matrix Multiplication (100×100)	25	200
2.	SVD (100×100 Matrix)	1500	49
3.	FFT on 2D image	2500	300
4.	Mandelbrot image (256 iterations)	6960	100
5.	image compression (8×8 discrete cosine xforms)	110	500

Table 7.3: Benchmarks for evaluating Warp performance.

REFERENCES

1. *VLSI Array Processors*

   S.Y.Kung, Prentice Hall 1988

2. H.T.Kung. *Why Systolic Architectures* ?

   IEEE Computer, Jan 1982

3. *Transputer Reference Manual*

   Prentice Hall 1988

4. *Transputer Development System*

   Prentice Hall 1988

5. *Occam 2 Reference Manual*

   Prentice Hall 1988

6. John Wexler and Dominic Prior. *Solving Problems With Transputers.*

   Microprocessors and Microsystems March 1989

7. Installation Guide And User Manual

8. *A Tutorial Introduction To Occam Programming*

   D.Pountain and D.May, Blachwell Scientific 1987

9. *Transputer Instruction Set: a compiler writer's guide*

   Prentice Hall 1988

10. *Systolic Signal Processing Systems*

    Marcel Dekker.Inc 1987

11. *Designing Efficient Algorithms For Parallel Computers*

    Michael j. Quinn. McGraw–Hill International 1987

12. *The Warp Computer: Architecture, Implementation,* and *Performance*

    Macro Annaratone et.al. IEEE Transactions on Computers, Dec 1987